

Java 虚拟机开发：

权衡优化、高效和安全的最优方案

介绍Java虚拟机开发的核心知识，内容深入、语言通俗易懂
全书精心筛选了虚拟机开发中最具代表性、最典型的知识点
采用了理论加实践的教学方法，兼顾理论、案例的完美展现

张善香 编著

清华大学出版社

解析 Java 虚拟机开发：权衡优化、高效和安全的最优方案

张善香　编　著

清华大学出版社
北　京

内容简介

本书细致分析了 Java 虚拟机开发的基本知识，为读者权衡出优化、高效和安全的最优方案。本书内容新颖、知识全面、讲解详细，全书共 17 章，第 1 章讲解一起走进 Java 世界的基本知识；第 2 章讲解 JDK 编译测试的基础知识；第 3 章讲解安全性考虑的核心知识；第 4 章讲解通过网络实现移动性的知识；第 5 章浅谈 Java 虚拟机内部机制的基础知识；第 6 章深入分析 class 文件的核心知识；第 7 章详细讲解栈和局部变量操作的知识；第 8 章深入详解内存异常和垃圾处理的基本知识；第 9 章讲解高效手段之性能监控工具和优化部署的核心知识；第 10 章讲解 JVM 参数分析和调优实战的知识；第 11 章讲解虚拟机类加载机制的基本知识；第 12 章讲解研究高效之魂；第 13 章讲解类加载器和执行子系统的基本知识；第 14 章讲解编译优化的基本知识；第 15 章讲解运行期优化的基本知识；第 16 章讲解内存模型和线程的基本知识；第 17 章讲解如何将安全和优化合二为一。全书内容循序渐进，并且逐一做到了深入剖析。

本书适合 Java 各个级别的程序员、研发人员及在职程序员，也可以作为相关培训学校和大专院校相关专业的教学用书。

图书在版编目(CIP)数据

解析 Java 虚拟机开发：权衡优化、高效和安全的最优方案/张善香编著. —北京：清华大学出版社，2013
ISBN 978-7-302-31494-3

Ⅰ. ①解… Ⅱ. ①张… Ⅲ. ①Java 语言—程序设计 Ⅳ. ①TP312

中国版本图书馆 CIP 数据核字(2013)第 024772 号

责任编辑：魏 莹
装帧设计：杨玉兰
责任校对：王 晖
责任印制：

出版发行：清华大学出版社
网 址：http://www.tup.com.cn, http://www.wqbook.com
地 址：北京清华大学学研大厦 A 座 邮 编：100084
社 总 机：010-62770175 邮 购：010-62786544
投稿与读者服务：010-62776969, c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015, zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn, 010-62791865
印 刷 者：
装 订 者：
经 销：全国新华书店
开 本：185mm×260mm **印 张：**31 **字 数：**751 千字
版 次：2013 年 6 月第 1 版 **印 次：**2013 年 6 月第 1 次印刷
印 数：1～4000
定 价：59.00 元

产品编号：

前　　言

自从 Java 语言自诞生以来，经过十多年的发展和应用，已经成为当今最流行的编程语言之一。根据某权威编程语言排行榜显示，它始终居于第一位。现在全球已有超过 15 亿部手机和手持设备应用 Java 技术。同时，Java 技术因其跨平台特性和良好的可移植性，成为广大软件开发技术人员的挚爱，是全球程序员的首选开发平台之一。

日益成熟的 Java 语言编程技术现在已无处不在，使用该编程技术可以进行桌面程序应用、Web 应用、分布式系统和嵌入式系统应用开发，并且在信息技术等各个领域得到广泛应用。

本书全面介绍了 Java 虚拟机技术的核心知识，全书内容深入并详解，作者用通俗的语言将大师级的知识展现在读者的眼前。

本书的内容

通过本书的内容，细致分析了 Java 虚拟机开发的基本知识，和读者们一起权衡出优化、高效和安全的最优方案。本书内容新颖、知识全面、讲解详细，全书分为 17 章，第 1 章讲解一起走进 Java 世界的基本知识；第 2 章讲解 JDK 编译测试的基础知识；第 3 章讲解安全性考虑的核心知识；第 4 章讲解通过网络实现移动性的知识；第 5 章浅谈 Java 虚拟机内部机制的基础知识；第 6 章深入分析 class 文件的核心知识；第 7 章详细讲解栈和局部变量操作的知识；第 8 章深入详解内存异常和垃圾处理的基本知识；第 9 章讲解高效手段之性能监控工具和优化部署的核心知识；第 10 章讲解 JVM 参数分析和调优实战的知识；第 11 章讲解虚拟机类加载机制的基本知识；第 12 章讲解研究高效之魂；第 13 章讲解类加载器和执行子系统的基本知识；第 14 章讲解编译优化的基本知识；第 15 章讲解运行期优化的基本知识；第 16 章讲解内存模型和线程的基本知识；第 17 章讲解如何将安全和优化合二为一。全书内容循序渐进，并且逐一做到了深入剖析。

本书特色

本书内容相当丰富，实例内容覆盖全面，满足 Java 程序员成长道路上的方方面面。我们的目标是通过一本图书，提供多本图书的价值，读者可以根据自己的需要有选择地阅读，以完善本人的知识和技能结构。在内容的编写上，本书具有以下特色。

(1) 专家写作，内容专业而深入

本书是国内一线著名的 Java 专家级作者的力作。为了本书确保广度和深度，本书并没有将大量篇幅用在规范和基本语法上，而是专注于各个基本知识的具体细节，尽量涉及了每个知识中最为重要的内容，并且讨论了相关的高级用法技术。

本书既是介绍性书籍，又是深入研究技术性书籍。本书实现了高级技术与介绍性知识并重的效果，为了达到这一目标，笔者做过大量研究。比如，参与论坛讨论，开发大量的实际项目，参加学术会议和研讨会，同时跟制定 Java 规范的专家组进行沟通，同全世界顶

级专家进行合作。

(2) 结构合理

从用户的实际需要出发，科学安排知识结构，内容由浅入深，叙述清楚，具有很强的知识性和实用性，反映了 Java 虚拟机的核心知识。同时全书精心筛选的最具代表性、读者最关心典型知识点，几乎包括虚拟机技术的各个方面。

(3) 易学易懂

本书条理清晰、语言简洁，可帮助读者快速掌握每个知识点；每个部分既相互连贯又自成体系，使读者既可以按照本书编排的章节顺序进行学习，也可以根据自己的需求对某一章节进行针对性的学习。

(4) 由浅入深

本书从 Java 语言的发展、开发环境及基本语法知识入手，逐步介绍了 JDK 编译测试、安全性、移动性、虚拟机的内部机制、class 文件、栈和局部变量操作、内存异常、垃圾处理、性能监控工具和优化部署、类的加载机制、类加载器和执行子系统、编译优化等知识。保证让读者在没有编程基础的情况下，也能够很快掌握 Java 虚拟机的各种技术。

(5) 实用性强

本书彻底摒弃枯燥的理论和简单的操作，注重实用性和可操作性，详细讲解了各个部分的源码知识，使用户掌握相关的操作技能的同时，还能学习到相应的基础知识。

读者对象

初学编程的自学者

编程爱好者

大中专院校的老师和学生

相关培训机构的老师和学员

毕业设计的学生

初中级程序开发人员

程序测试及维护人员

参加实习的初级级程序员

在职程序员

资深程序员

本团队在编写编写过程中，得到了清华大学出版社工作人员的大力支持。但是本团队水平毕竟有限，如有纰漏和不尽如人意之处在所难免，诚请读者提出意见或建议，以便修订并使之更臻完善。另外，为了方便读者学习，我们特开通了技术支持 QQ 群，群号为 75593028，欢迎读者加入本群。

编　者

目　录





Java

第 1 章

一起走进 Java 世界

几乎所有学习计算机的朋友都听说过 Java 语言，Java 语言“犀利无比”，已占据了当前软件应用的半壁江山。本章将首先带领读者领略 Java 这门强大的语言，仔细品味它的每一个强大之处，并且详细分析 Java 技术体系的构成，为 Java 程序员规划发展之路。

1.1 Java 的优势

Java 语言在编程语言中占据了重要地位，用 Java 开发的项目分布在世界各地的各个领域。为了让读者学好 Java 语言更有信心，本节首先介绍学习 Java 语言的优势。

1.1.1 排名第一的编程语言

都说 Java 语言是使用率最高的一门编程语言，并且没有之一。表 1-1 是最新的编程语言排行榜，截止至 2012 年 2 月。

表 1-1 编程语言排行榜(截止至 2012 年 2 月)

排 名	语 言	使用率/%
1	Java	17.050
2	C	16.523
3	C#	8.653
4	C++	7.853
5	Object-C	7.062
6	PHP	5.641

1.1.2 提供给我们美好的就业前景

由表 1-1 的统计数据可以看出，Java 语言是当今使用率最高的编程语言。使用率如此之高，也决定了 Java 程序员的就业机会要大于其他开发者。

Java 的功能比较强大，在服务器领域、移动设备、桌面应用和 Web 领域都占据了重要的地位。

- 服务器领域：Java 在服务器编程方面很强悍，拥有很多其他语言所没有的优势。
- 移动设备：Java 在手机领域的应用比较广泛，手机 Java 游戏随处可见，当前异常火爆的移动开发平台——Android，上面的应用项目基本上都是用 Java 开发的。
- 桌面应用：Java 和 C++、.NET 一样重要，影响着桌面程序的发展。
- Web 领域：Java Web 有着巨大的优势，无论是开发工具还是开发框架都是开源的，并且安全性更强。

正是因为 Java 语言可以在多个领域开发项目，所以市面上需要多个领域的 Java 程序员。据统计，当前全球有 30 亿 Java 器件正在运行着 Java 程序，500 多万 Java 开发者活跃在地球的每个角落，数以千万计的 Web 用户每次上网都亲历 Java 的威力。今天，Java 运行在 9.08 亿手机、10 亿智能卡和 10 亿 PC 上，并为 28 款可兼容的应用服务器提供了功能强大的平台。这么多应用，彻底改变了用户的生活。越来越多的企业，因为使用了 Java 而提高了生产效率。在中国，越来越多的用户，因为 Java 而降低了成本，享受了生活。

作为唯一在互联网上开发的语言，Java 平台以其移动性、安全性和开放性受到追捧。据

IDC 预计，自 2010 年起的其后 5 年内，采用 Java 的 IT 产品的价值将翻番，在 2015 年将达到 45.53 亿美元，年增长率为 14.9%。截止到 2010 年 5 月，Java 的注册开发商超过 1300 万人，对 JRE(Java 运行环境)的下载达 9200 万次。詹姆斯·戈士林博士预计在 3~5 年内 Java 技术开发商将发展到 2000 万，无线 Java 也在迅速攀升。

上述发展趋势在国内更是如此，当前我国对软件人才的需求已达 50 万，并且以每年 20%左右的速度增长。在未来 5 年内，合格软件人才的需求将远大于供给。在过去的 2011 年，我国软件人才的缺口已达 52.5 万人，其中尤以 Java 人才最为缺乏。

根据 IDC 的统计数字，在所有软件开发类人才的需求中，对 Java 工程师的需求达到全部需求量的 60%~70%。这也就不难解释在智联招聘和 51job 等主流招聘网站，随处可见 Java 工程师的招聘广告了。

1.2 学习 Java 需要了解的那些事

了解了学习 Java 语言的优势之后，接下来需要了解和这门神奇语言相关的几件大事，下面介绍这门神奇语言的发展历史和特点等，为读者学习本书后面的知识打下理论基础。

1.2.1 品 Java 语言的发展历史

Java 是由 Sun Microsystems 公司于 1995 年 5 月推出的 Java 程序设计语言(以下简称 Java 语言)和 Java 平台的总称。用 Java 实现的 HotJava 浏览器(支持 Java Applet)向我们展示了 Java 语言的魅力——跨平台、动态的 Web、Internet 计算。从那以后，Java 便被广大程序员和企业用户广泛接受，成为了当今最受欢迎的编程语言之一。

Java 平台由 Java 虚拟机(Java Virtual Machine)和 Java 应用编程接口(Application Programming Interface，API)构成。Java 应用编程接口为 Java 应用提供了一个独立于操作系统的标准接口，可分为基本部分和扩展部分。在硬件或操作系统平台上安装一个 Java 平台之后，Java 应用程序就可运行。现在 Java 平台已经嵌入了几乎所有的操作系统。这样 Java 程序可以只编译一次，就在各种系统中运行。Java 应用编程接口已经从 1.1x 版发展到 1.2 版。目前常用的 Java 平台基于 Java 1.6，最新版本为 Java 1.7。

当 1995 年 Sun 公司推出 Java 语言之后，全世界的目光都被这个神奇的语言所吸引。那么 Java 到底有何神奇之处呢？Java 语言其实最早诞生于 1991 年，起初被称为 OAK 语言，是 Sun 公司为一些消费性电子产品而设计的一个通用环境。Sun 公司的最初目的是为了开发一种独立于平台的软件技术，而且在网络出现之前 OAK 是默默无闻的，甚至差一点夭折。但是随着网络的出现彻底改变了 OAK 的命运。在 Java 出现以前，Internet 上的信息都是一些乏味死板的 HTML 文档。这对于那些迷恋于 Web 浏览的人们来说简直不可容忍。他们迫切希望能在 Web 中看到一些交互式的内容，开发人员也极希望能够在 Web 上创建一类无需考虑软硬件平台就可以执行的应用程序，当然这些程序还要有极大的安全保障。对于用户的这种要求，传统的编程语言显得无能为力。Sun 的工程师敏锐地察觉到了

这一点，从 1994 年起，他们开始将 OAK 技术应用于 Web 上，并且开发出了 HotJava 的第一个版本。并最终在 1995 年，将 Java 技术展现在了世人的面前。

Java 分为以下三个体系。

- JavaSE：是 Java 2 Platform Standard Edition 的缩写，即 Java 平台标准版。
- JavaEE：是 Java 2 Platform Enterprise Edition 的缩写，即 Java 平台企业版。
- JavaME：是 Java 2 Platform Micro Edition 的缩写，即 Java 平台微型版。

2009 年 4 月 20 日，Oracle(甲骨文)宣布成功收购 Sun 公司，从那以后，Java 成为了软件巨头甲骨文旗下的一款产品，并和 Oracle 数据库一起推动着世界科技继续向前发展。

1.2.2 Java 的特点

- 面向对象：Java 语言提供了类、接口和继承等特性。为了简单起见，Java 只支持类之间的单继承和接口之间的多继承，并且也支持类与接口之间的实现机制。总之，Java 语言是一门纯粹面向对象的程序设计语言。
- 简单：Java 语言的语法与 C 语言和 C++语言十分接近，这样大多数程序员可以很容易地学习和使用 Java。并且 Java 抛弃了 C++中的操作符重载、多继承、自动强制类型和指针等知识，这将更加利于我们学习并掌握它。另外，Java 还提供了自动废料收集机制，使得程序员不必再为内存管理而担忧。
- 分布式：Java 语言支持 Internet 应用开发，在基本的 Java 应用编程接口中有一个网络应用编程接口(java.net)，通过这个接口提供了用于网络应用编程的类库，包括 URL、URLConnection、Socket、ServerSocket 等。Java 的 RMI(远程方法激活)机制也是开发分布式应用的重要手段。
- 健壮：Java 的强类型机制、异常处理、废料的自动收集等是 Java 程序健壮性的重要保证。Java 通过安全检查机制，使 Java 程序更具健壮性。
- 可移植：移植性是指能够在不同的开发平台和服务器平台上使用。因为 Java 的运行环境是用 ANSI C 实现的，所以 Java 系统本身具有很强的可移植性，可以在很多平台上运行。无论是微软的产品还是 IBM 的产品，都可以运行 Java 程序。
- 高性能：与解释型的高级脚本语言相比，Java 的确是高性能的。随着 JIT(Just-In-Time)编译器技术的发展，Java 的运行速度已经越来越接近于 C++。
- 多线程：当程序需要同时处理多项任务时就需要多线程开发，一个程序在同一时间只能做一件事情的功能过于简单，肯定无法满足现实的需求。在实际的应用中，多线程开发是必不可少的，多线程的目的是在同一时间可以做多件事情。并且可以开启多个线程同时做一件事情，这样可以提高效率。不管是 C 语言、C++还是其他的程序设计语言，线程都是一个十分重要的知识点，多线程是现代开发软件系统的发展方向，Java 作为当今的主流程序设计，它当然是支持多线程的，具有并发性，其执行的效率很高。
- 动态：Java 语言的设计目标之一是适应于动态变化的环境。Java 程序中的类需要动态地被载入到运行环境中，也可以通过网络来载入所需要的类。动态语言的好

处是有利于软件升级。另外，Java 中的类有一个运行时刻的表示，能进行运行时刻的类型检查。

1.3　剖析 Java 的运行机制

Java 语言是一门高级语言，它既有解释型语言的特性，也具有编译语言的特性。Java 需要先编译，然后再解释运行。本节将简要介绍 Java 语言的运行机制，更加深入地介绍这门神奇的语言。

1.3.1　高级语言的运行机制

高级语言按照执行方式可以分为解释型和编译型两种。

1) 解释型语言

计算机不能直接理解高级语言，只能直接理解机器语言，所以必须要把高级语言翻译成机器语言，计算机才能执行高级语言编写的程序。

翻译的方式有两种，一种是编译，一种是解释。

解释型语言的程序不需要编译，省了道工序，在运行程序的时候才翻译，比如解释型 Basic 语言，专门有一个解释器能够直接执行 Basic 程序，每个语句都是执行的时候才翻译。这样解释型语言每执行一次就要翻译一次，效率比较低，是一句一句地翻译。

2) 编译型语言

编译型语言写的程序执行之前，需要一个专门的编译过程，把程序编译成为机器语言的文件，比如 exe 文件，以后要运行的话就不用重新翻译了，直接使用编译的结果就行了(exe 文件)，因为翻译只做了一次，运行时不需要翻译，所以编译型语言的程序执行效率高。

编译型与解释型，两者各有利弊。前者由于程序执行速度快，同等条件下对系统要求较低，因此像开发操作系统、大型应用程序、数据库系统等时都采用它，像 C/C++、Pascal/Object Pascal(Delphi)等都是编译语言，而一些网页脚本、服务器脚本及辅助开发接口这样的对速度要求不高、对不同系统平台间的兼容性有一定要求的程序则通常使用解释型语言，如 Java、JavaScript、VBScript、Perl、Python、Ruby、MATLAB 等。

但随着硬件的升级和设计思想的变革，编译型和解释型语言越来越笼统，主要体现在一些新兴的高级语言上，而解释型语言的自身特点也使得编译器厂商愿意花费更多成本来优化解释器，解释型语言性能超过编译型语言也是必然的。

1.3.2　Java 的运行机制

Java 应用程序的开发周期包括编译、下载、解释和执行几个部分。Java 编译程序将 Java 源程序翻译为 JVM 可执行代码——字节码。这一编译过程同 C/C++的编译有些不同。当 C 编译器编译生成一个对象的代码时，该代码是为在某一特定硬件平台运行而产生

的。因此在编译过程中，编译程序通过查表将所有对符号的引用转换为特定的内存偏移量，目的是保证程序运行。Java 编译器不将对变量和方法的引用编译为数值引用，也不确定程序执行过程中的内存布局，而是将这些符号引用信息保留在字节码中，由解释器在运行过程中创立内存布局，然后再通过查表来确定一个方法所在的地址。这样就有效地保证了 Java 的可移植性和安全性。

运行 JVM 字节码的工作是由解释器(Java 命令)来完成的。整个解释执行过程分为以下三步进行。

- 代码的装入。
- 代码的校验。
- 代码的执行。

装入代码的工作由“类装载器”(Class Loader)完成。类装载器负责装入运行一个程序需要的所有代码，这也包括程序代码中的类所继承的类和被其调用的类。当类装载器装入一个类时，该类被放在自己的名字空间中。除了通过符号引用自己名字空间以外的类，类之间没有其他办法可以影响其他类。在本台计算机上的所有类都在同一地址空间内，而所有从外部引进的类，都有一个自己独立的名字空间。这使得本地类通过共享相同的名字空间获得较高的运行效率，同时又保证它们与从外部引进的类不会相互影响。当装入了运行程序需要的所有类后，解释器便可确定整个可执行程序的内存布局。解释器为符号引用同特定的地址空间建立对应关系及查询表。通过在这一阶段确定代码的内存布局，Java 很好地解决了由超类改变而使子类崩溃的问题，同时也防止了代码对地址的非法访问。

接下来，被装入的代码由字节码校验器进行检查。校验器可发现操作数栈溢出、非法数据类型转化等多种错误。通过校验后，代码便开始执行了。

有如下两种 Java 字节码的执行方式。

(1) 即时编译方式：解释器先将字节码编译成机器码，然后再执行该机器码。

(2) 解释执行方式：解释器通过每次解释并执行一小段代码来完成 Java 字节码程序的所有操作。

在 Java 中通常采用的是第二种方法，因为 JVM 规格描述具有足够的灵活性，这使得将字节码翻译为机器代码的工作具有较高的效率。对于那些对运行速度要求较高的应用程序，解释器可将 Java 字节码即时编译为机器码，从而很好地保证了 Java 代码的可移植性和高性能。

Java 程序的运行必须经过编写、编译、运行三个步骤。

- 编写是指在 Java 开发环境中进行程序代码的输入，最终形成后缀名为.java 的 Java 源文件。
- 编译是指使用 Java 编译器对源文件进行错误排查的过程，编译后将生成后缀名为.class 的字节码文件，这不像 C 语言那样最终生成可执行文件。
- 运行是指使用 Java 解释器将字节码文件翻译成机器代码，执行并显示结果。这一过程如图 1-1 所示。



图 1-1　Java 程序的运行流程

在图 1-1 中，字节码文件是一种和任何具体机器环境及操作系统环境无关的中间代码，它是一种二进制文件，是 Java 源文件由 Java 编译器编译后生成的目标代码文件。编程人员和计算机都无法直接读懂字节码文件，它必须由专用的 Java 解释器来解释执行，因此 Java 是一种在编译基础上进行解释运行的语言。

Java 解释器负责将字节码文件翻译成具体硬件环境和操作系统平台下的机器代码，以便执行。因此 Java 程序不能直接运行在现有的操作系统平台上，它必须运行在被称为 Java 虚拟机的软件平台之上。

1.3.3　Java 虚拟机——JVM

JVM 是一种用于计算设备的规范，可用不同的方式(软件或硬件)加以实现。编译虚拟机的指令集与编译微处理器的指令集非常类似。Java 虚拟机包括一套字节码指令集、一组寄存器、一个栈、一个垃圾回收堆和一个存储方法域。

Java 虚拟机(JVM)是可运行 Java 代码的假想计算机。只要根据 JVM 规格描述将解释器移植到特定的计算机上，就能保证经过编译的任何 Java 代码能够在该系统上运行。

1) 为什么要使用 JVM

Java 语言的一个非常重要的特点就是与平台的无关性，而使用 Java 虚拟机是实现这一特点的关键。一般的高级语言如果要在不同的平台上运行，至少需要编译成不同的目标代码。而引入 Java 语言虚拟机后，Java 语言在不同平台上运行时不需要重新编译。Java 语言使用模式 Java 虚拟机屏蔽了与具体平台相关的信息，使 Java 语言编译程序只需生成在 Java 虚拟机上运行的目标代码(字节码)，就可以在多种平台上不加修改地运行。Java 虚拟机在执行字节码时,把字节码解释成具体平台上的机器指令执行。

2) JVM 的作用

Java 虚拟机(JVM)是运行 Java 程序的软件环境，Java 解释器就是 Java 虚拟机的一部分。在运行 Java 程序时，首先会启动 JVM，然后由它来负责解释执行 Java 的字节码，并且 Java 字节码只能运行于 JVM 之上。这样利用 JVM 就可以把 Java 字节码程序和具体的硬件平台以及操作系统环境分隔开来，只要在不同的计算机上安装了针对于特定具体平台的 JVM，Java 程序就可以运行，而不用考虑当前具体的硬件平台及操作系统环境，也不用考虑字节码文件是在何种平台上生成的。JVM 把这种不同软硬件平台的具体差别隐藏起来，从而实现了真正的二进制代码级的跨平台移植。JVM 是 Java 平台无关的基础，Java 的跨平台特性正是通过在 JVM 中运行 Java 程序实现的。Java 的这种运行机制可以通过图 1-2 来说明。

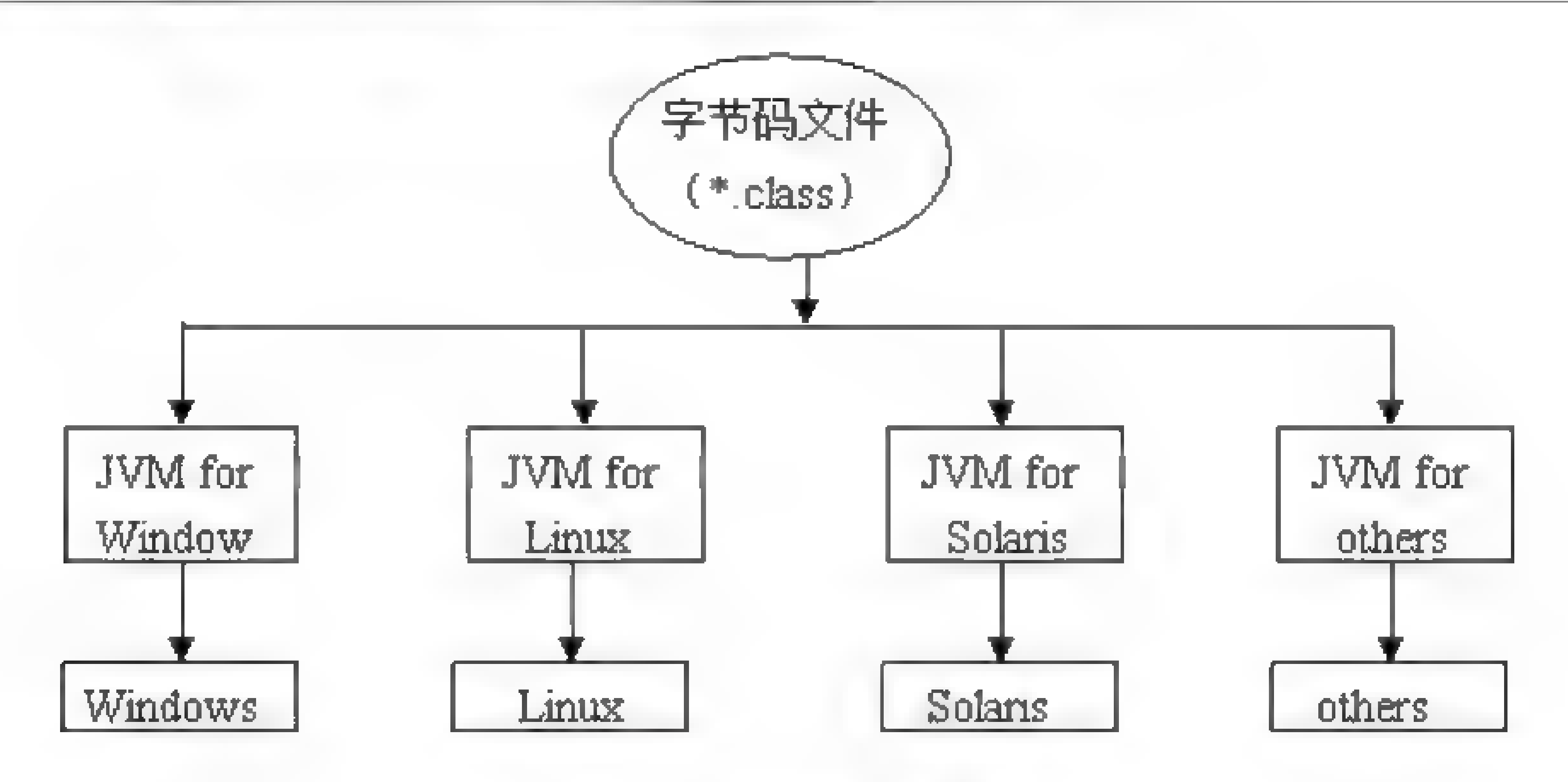


图 1-2 JVM 的工作方式

Java 语言这种“一次编写，到处运行(Write once，run anywhere)”的方式，有效地解决了目前大多数高级程序设计语言需要针对不同系统来编译产生不同机器代码的问题，即硬件环境和操作平台的异构问题，大大降低了程序开发、维护和管理的开销。

需要注意的是，Java 程序通过 JVM 可以达到跨平台运行，但 JVM 是不跨平台的。也就是说，不同操作系统上的 JVM 是不同的，Windows 平台之上的 JVM 不能用在 Linux 上面，反之亦然。

1.3.4 独特的垃圾回收机制

在 Java 的众多突出特性之中，垃圾回收机制是最具代表性的一个。在 C++中，对象所占的内存在程序结束运行之前一直被占用，在明确释放之前不能分配给其他对象；而在 Java 中，当没有对象引用指向原先分配给某个对象的内存时，该内存便成为垃圾。JVM 的一个系统级线程会自动释放该内存块。垃圾收集意味着程序不再需要的对象是“无用信息”，这些信息将被丢弃。当一个对象不再被引用的时候，内存回收它占领的空间，以便空间被后来的新对象使用。事实上，除了释放没用的对象，垃圾收集也可以清除内存记录碎片。由于创建对象和垃圾收集器释放丢弃对象所占的内存空间，内存会出现碎片。碎片是分配给对象内存块之间的空闲内存洞。碎片整理将所占用的堆内存移到堆的一端，JVM 将整理出的内存分配给新的对象。

垃圾收集能自动释放内存空间，减轻编程的负担，这使 Java 虚拟机具有一些优点。首先，它能使编程效率提高。在没有垃圾收集机制的时候，可能要花许多时间来解决一个难懂的存储器问题。在用 Java 语言编程的时候，其垃圾收集机制可大大节省时间。其次是它能保护程序的完整性，垃圾收集是 Java 语言安全性策略的一个重要部分。

垃圾收集的一个潜在的缺点是它的开销影响程序性能。Java 虚拟机必须追踪运行程序中有用的对象，而且最终释放没用的对象。这一过程需要花费处理器的时间。其次是垃圾收集算法的不完备性，早先采用的某些垃圾收集算法就不能保证 100%收集到所有的废弃内存。当然随着垃圾收集算法的不断改进以及软硬件运行效率的不断提升，这些问题都可以迎刃而解。

1.4　剖析 Java 语言体系

Java 语言博大精深，是现实应用中使用最多的一门编程语言。Java 语言的应用范围之广，作为一名学习者，或者一名程序员，只能循序渐进地从基础开始学习，或者专门向一个分支“倾斜”发展。只要读者明确自己将向哪一方面发展，那么接下来的内容便可以帮助大家明确学习什么内容才能实现自己的目标。

1.4.1　Java 程序员的 6 个级别

所谓 6 个级别，只是笔者的观点，其实并没有权威性，但是有较强的代表性。

1．第一个级别——普通 Java 程序员

需要学习并掌握下面的内容。

1) Java 开发环境

JDK、JVM、Eclipse、Linux。

2) Java 核心编程技术

学习 Java 必须从 Java 开发环境开始，到 Java 语法，再到 Java 的核心 API。

- Java 开发入门：Java 开发环境的安装与使用，包括 JDK 命令、Eclipse IDE、Linux 下 Java 程序的开发和部署等。
- Java 语法基础：基于 JDK 和 Eclipse 环境，进行 Java 核心功能开发，掌握 Java 面向对象的语法构成，包括类、抽象类、接口、最终类、静态类、匿名类、内部类、异常的编写。
- Java 核心 API：基于 JDK 提供的类库，掌握三大核心功能：
  - Java 核心编程：包括 Java 编程的两大核心功能——Java 输入/输出流和多线程，以及常用的辅助类库——实体类、集合类、正则表达式、XML 和属性文件。
  - Java 图形编程：包括 Sun 的 GUI 库 AWT(Java2D、JavaSound、JMF)和 Swing、IBM 和 GUI 库 SWT 和 Jface;
  - Java 网络编程：Applet 组件编程、Socket 编程、NIO 非阻塞 Socket 编程、RMI 和 CORBA 分布式开发。

3) 核心编程

IO、多线程、实体类、 集合类、正则表达式、XML 和属性文件。

4) 图形编程

AWT(Java2D/JavaSound/JMF)、Swing、SWT、JFace。

5) 网络编程

Applet、Socket/TCP/UDP、NIO、RMI、CORBA。

6) 高级特性

反射、泛型、注释符、自动装箱和拆箱、枚举类、可变参数、可变返回类型、增强循

环、静态导入。

2. 第二个级别——JavaEE 初级软件工程师

需要学习并掌握下面的内容。

1) JSF 框架开发技术

配置文件(页面导航、后台 Bean)、JSF 组件库(JSF EL 语言、HTML 标签、事件处理)、JSF 核心库(格式转换、输入验证、国际化)。

2) JavaWeb 核心开发技术

- 开发环境(Eclipse、Linux)。
- 三大组件(JSP、JavaBean、Servlet)。
- 扩展技术(EL、JSTL、Taglib)。
- 网页开发技术(HTML、XML、CSS、JavaScript、AJAX)。
- 数据库设计技术(SQL、MySql、Oracle、SQLServer、JDBC)。

3) Web 服务器

常用的服务器主要有 Tomcat、Jetty、Resin 和 JbossWeb。

3. 第三个级别——JavaEE 中级软件工程师

需要学习并掌握下面的内容。

1) Struts1 表现层框架

入门配置、核心组件、标签库、国际化、数据校验、数据库开发、Sitemesh 集成、集成 Hibernate/iBATIS。

2) Struts2 表现层框架

入门配置、核心组件、标签库、国际化、数据校验、Sitemesh 集成转换器、拦截器、集成 Hibernate/iBATIS。

3) Spring 业务层框架

入门配置、IoC 容器、MVC、标签库、国际化、数据校验、数据库开发。

4) Hibernate 持久层框架

MySQL、Oracle、SQL Server。

5) iBATIS 持久层框架

MySQL、Oracle、SQL Server。

6) Web 服务器

Tomcat、Jetty、Resin、JBossWeb。

4. 第四个级别——Java 高级软件工程师

需要学习并掌握下面的内容。

(1) JavaWeb 开源技术与框架。

(2) JavaWeb 分布式开发技术。

(3) JTA(Java 事物管理)。

(4) JAAS(Java 验证和授权服务)。

(5) JNDI(Java 命名和目录服务)。
(6) JavaMail(Java 邮件服务)。
(7) JMS(Java 信息服务)、WebService(Web 服务)。
(8) JCA(Java 连接体系)、JMS(Java 管理体系)。
(9) 应用服务器(JbossAS、WebLogic、WebSphere)。

5．第五个级别——JavaEE 系统架构师

需要学习并掌握下面的内容。

1) 面向云架构(COA)

COA、SaaS、网格计算、集群计算、分布式计算、云计算。

2) 面向资源架构(ROA)

ROA、RESI。

3) 面向 Web 服务架构(SOA)

WebService、SOA、SCA、ESB、OSGI、EAI。

4) Java 设计模式

- 创建式模式：抽象工厂、建造者、工厂方法、原型、单例。
- 构造型模式：适配器、桥接、组合、装饰、外观、享元、代理。
- 行为型模式：责任链、命令、解释器、迭代子、中介者、备忘录、观察者、状态、策略、模板方法、访问者。

5) Java 与 UML 建模

对象图、用例图、组件图、部署图、序列图、交互图、活动图、正向工程与逆向工程。

6．第六个级别——CTO 首席技术官

需要学习并掌握下面的内容。

- 发展战略。
- 技术总监。
- 团队提升。
- 团队建设。
- 项目管理。
- 产品管理。

1.4.2　分析 Java 体系的构成

从传统意义上来看，Sun 官方所定义的 Java 技术体系主要包括以下 5 个组成部分。

- Java 程序设计语言。
- 各种硬件平台上的 Java 虚拟机。
- Class 文件格式。
- Java API 类库。
- 来自商业机构和开源社区的第三方 Java 类库。

通常把 Java 程序设计语言、Java 虚拟机和 Java API 类库统称为 Java Development Kit，简称 JDK。JDK 是支持 Java 程序开发的最小环境。另外，可以把 Java API 类库中的 Java SE API 子集和 Java 虚拟机这两部分统称为 JRE (Java Runtime Environment)，JRE 是支持 Java 程序运行的标准环境。

上述组成是根据每个部分的功能进行划分的，如果按照 Java 技术关注的重点业务领域来划分，Java 技术体系可以分为如下 4 个平台。

- Java Card：支持一些 Java 小程序(Applets)运行在小内存设备(如智能卡)上的平台。
- Java ME(Micro Edition)：支持 Java 程序运行在移动终端(手机、PDA)上的平台，对 Java API 有所精简，并加入了针对移动终端的支持，这个版本以前称为 J2ME。
- Java SE (Standard Edition)：支持面向桌面级应用(如 Windows 下的应用程序)的 Java 平台，提供了完整的 Java 核心 API，这个版本以前称为 J2SE。
- Java EE(Enterprise Edition)：是 J2EE 的升级，支持使用多层架构企业应用(如 ERP、CRM 应用)的 Java 平台，除了提供 Java SE API 外，还对其做了大量的扩充并提供了相关的部署支持。

1.5 Java 虚拟机家族

本书的主角是虚拟机(Virtual Machine)，虚拟机是指通过软件模拟的具有完整硬件系统功能的、运行在一个完全隔离环境中的完整计算机系统。本节将简要介绍 Java 虚拟机的基本知识。

1.5.1 虚拟机的用途

在现实应用中，对于一般计算机用户来说，最常见的使用虚拟机的情形是安装双系统。例如，在 Windows 平台上安装一个虚拟机，然后在这个虚拟机中安装 Linux 操作系统或 iOS 系统，这样就实现了双系统功能。

正如上面描述的那样，通过虚拟机可以在一台物理计算机上模拟出一台或多台虚拟的计算机，这些虚拟机就像真正的计算机那样进行工作，例如，你可以安装操作系统、安装应用程序、访问网络资源等。对于用户而言，它只是运行在用户物理计算机上的一个应用程序，但是对于在虚拟机中运行的应用程序而言，它就是一台真正的计算机。因此，当在虚拟机中进行软件评测时，可能系统一样会崩溃，但是崩溃的只是虚拟机上的操作系统，而不是物理计算机上的操作系统，并且使用虚拟机的 Undo(恢复)功能，可以马上恢复虚拟机到安装软件之前的状态。

1.5.2 理解 Java 虚拟机

JVM(Java 虚拟机)是 Java Virtual Machine 的缩写，它是一个虚构出来的计算机，是通

过在实际的计算机上仿真模拟各种计算机功能来实现的。Java 虚拟机有自己完善的硬件架构，如处理器、堆栈、寄存器等，还具有相应的指令系统。

1．为什么要使用 Java 虚拟机

Java 语言的一个非常重要的特点就是与平台的无关性，而使用 Java 虚拟机是实现这一特点的关键。一般的高级语言如果要在不同的平台上运行，至少需要编译成不同的目标代码。而引入：Java 语言虚拟机后，Java 语言在不同平台上运行时不需要重新编译。Java 语言使用模式：Java 虚拟机屏蔽了与具体平台相关的信息，使得 Java 语言编译程序只需生成在 Java 虚拟机上运行的目标代码(字节码)，就可以在多种平台上不加修改地运行。Java 虚拟机在执行字节码时，把字节码解释成具体平台上的机器指令执行。

2．谁需要了解 Java 虚拟机

Java 虚拟机是 Java 语言底层实现的基础，对 Java 语言感兴趣的人都应对 Java 虚拟机有个大概的了解。这有助于理解 Java 语言的一些性质，也有助于使用 Java 语言。对于要在特定平台上实现 Java 虚拟机的软件人员，Java 语言的编译器作者以及要用硬件芯片实现 Java 虚拟机的人来说，则必须深刻理解 Java 虚拟机的规范。另外，如果想扩展 Java 语言，或是把其他语言编译成 Java 语言的字节码，也需要深入地了解 Java 虚拟机。

1.5.3　Java 虚拟机的数据类型

Java 虚拟机可以支持的 Java 语言的基本数据类型如下。

- byte：1 字节，有符号整数的补码。
- short：2 字节，有符号整数的补码。
- int：4 字节，有符号整数的补码。
- long：8 字节，有符号整数的补码。
- float：4 字节，IEEE754 单精度浮点数。
- double：8 字节，IEEE754 双精度浮点数。
- char：2 字节，无符号 Unicode 字符。
- object：对一个 Javaobject(对象)的 4 字节引用。
- returnAddress：4 字节，用于 jsr/ret/jsr-w/ret-w 指令。

几乎所有的 Java 类型检查都是在编译时完成的，上述列出的原始数据类型的数据在 Java 执行时不需要用硬件标记。操作这些原始数据类型数据的字节码(指令)本身就已经指出了操作数的数据类型，例如，iadd、ladd、fadd 和 dadd 指令都是把两个数相加，其操作数类型分别是 int、long、float 和 double。虚拟机没有给 boolean(布尔)类型设置单独的指令。boolean 型的数据是由 integer 指令，包括 integer 返回来处理的。boolean 型的数组则是用 byte 数组来处理的。虚拟机使用 IEEE754 格式的浮点数，不支持 IEEE 格式的较旧的计算机，在运行 Java 数值计算程序时可能会非常慢。

虚拟机的规范对于 object 内部的结构没有任何特殊的要求。在 Sun 公司的实现中，对 object 的引用是一个句柄，其中包含一对指针：一个指针指向该 object 的方法表，另一个指向该 object 的数据。用 Java 虚拟机的字节码表示的程序应该遵守类型规定。Java 虚拟机

的实现应拒绝执行违反了类型规定的字节码程序。Java 虚拟机由于字节码定义的限制似乎只能运行在 32 位地址空间的机器上。但是可以创建一个 Java 虚拟机，它自动地把字节码转换成 64 位的形式。从 Java 虚拟机支持的数据类型可以看出，Java 对数据类型的内部格式进行了严格规定，这样使得各种 Java 虚拟机的实现对数据的解释是相同的，从而保证了 Java 的与平台无关性和可移植性。

1.5.4 Java 虚拟机体系结构

Java 虚拟机由 5 个部分组成：一组指令集、一组寄存器、一个栈、一个无用单元收集堆(Garbage-collected-heap)和一个方法区域。这五部分是 Java 虚拟机的逻辑成分，不依赖任何实现技术或组织方式，但它们的功能必须在真实机器上以某种方式实现。接下来将简要介绍上述组成部分的基本知识，更加详细的内容请参阅本书后面的内容。

1. Java 指令集

Java 虚拟机支持大约 248 个字节码，每个字节码执行一种基本的 CPU 运算，例如，把一个整数加到寄存器、子程序转移等。Java 指令集相当于 Java 程序的汇编语言。

Java 指令集中的指令包含一个单字节的操作符，用于指定要执行的操作，还有 0 个或多个操作数，提供操作所需的参数或数据。许多指令没有操作数，仅由一个单字节的操作符构成。

虚拟机内层循环的执行过程如下：

```
do{
取一个操作符字节;
根据操作符的值执行一个动作;
}while(程序未结束)
```

因为指令系统的简单性，所以使得虚拟机执行的过程十分简单，这样有利于提高执行的效率。指令中操作数的数量和大小是由操作符决定的。如果操作数比一个字节大，那么它的存储顺序是高位字节优先。假如一个 16 位的参数存放时占用两个字节，其值为：

第一个字节*256+第二个字节

字节码指令流一般只是字节对齐的，但是指令 tabltch 和 lookup 是例外，在这两条指令内部强制要求的 4 字节边界对齐。

2. 寄存器

Java 虚拟机的寄存器用于保存机器的运行状态，与微处理器中的某些专用寄存器类似，所有寄存器都是 32 位的。在 Java 虚拟机中有如下 4 种寄存器。

- pc：Java 程序计数器。
- optop：指向操作数栈顶端的指针。
- frame：指向当前执行方法的执行环境的指针。
- vars：指向当前执行方法的局部变量区第一个变量的指针。

Java 虚拟机是栈式的，它不定义或使用寄存器来传递或接受参数，其目的是为了保证指令集的简洁性和实现时的高效性，特别是对于寄存器数目不多的处理器。

3. 栈

Java 虚拟机中的栈有三个区域，分别是局部变量区、运行环境区、操作数栈区。

1) 局部变量区

每个 Java 方法使用一个固定大小的局部变量集。它们按照与 vars 寄存器的字偏移量来寻址。局部变量都是 32 位的。长整数和双精度浮点数占据了两个局部变量的空间，却按照第一个局部变量的索引来寻址(例如：一个具有索引 n 的局部变量，如果是一个双精度浮点数，那么它实际占据了索引 n 和 n+1 所代表的存储空间)。虚拟机规范并不要求在局部变量中 64 位的值是 64 位对齐的。虚拟机提供了把局部变量中的值装载到操作数栈的指令，也提供了把操作数栈中的值写入局部变量的指令。

2) 运行环境区

在运行环境中包含的信息可以实现动态链接、正常的方法返回、异常和错误传播。

(1) 动态链接

运行环境包括对指向当前类和当前方法的解释器符号表的指针，用于支持方法代码的动态链接。方法的 class 文件代码在引用要调用的方法和要访问的变量时使用符号。动态链接把符号形式的方法调用翻译成实际方法调用，装载必要的类以解释还没有定义的符号，并把变量访问翻译成与这些变量运行时的存储结构相应的偏移地址。动态链接方法和变量使得方法中使用的其他类的变化不会影响到本程序的代码。

(2) 正常的方法返回

如果当前方法正常地结束了，在执行了一条具有正确类型的返回指令时，调用的方法会得到一个返回值。执行环境在正常返回的情况下用于恢复调用者的寄存器，并把调用者的程序计数器增加一个恰当的数值，以跳过已执行过的方法调用指令，然后在调用者的执行环境中继续执行下去。

(3) 异常和错误传播

异常情况在 Java 中被称作 Error(错误)或 Exception(异常)，是 Throwable 类的子类，在程序中出现的原因有以下两点：

- 动态链接错，如无法找到所需的 class 文件。
- 运行时出错，如对一个空指针的引用程序使用了 throw 语句。当发生异常时，Java 虚拟机采取如下措施解决。
  - 检查与当前方法相联系的 catch 子句表。每个 catch 子句包含其有效指令范围、能够处理的异常类型，以及处理异常的代码块地址。
  - 与异常相匹配的 catch 子句应该符合下面的条件：造成异常的指令在其指令范围之内，发生的异常类型是其能处理的异常类型的子类型。如果找到了匹配的 catch 子句，那么系统转移到指定的异常处理块处执行。如果没有找到异常处理块，会重复寻找匹配的 catch 子句的过程，直到当前方法所有嵌套的 catch 子句都被检查过。
  - 由于虚拟机从第一个匹配的 catch 子句处继续执行，所以 catch 子句表中的顺序是很重要的。因为 Java 代码是结构化的，因此总可以把某个方法的所有异常处理器都按序排列到一个表中，对任意可能的程序计数器的值，都可以用线

性的顺序找到合适的异常处理块，以处理在该程序计数器值下发生的异常情况。

- 如果找不到匹配的 catch 子句，那么当前方法得到一个“未截获异常”的结果并返回到当前方法的调用者，好像异常刚刚在其调用者中发生一样。如果在调用者中仍然没有找到相应的异常处理块，那么这种错误传播将被继续下去。如果错误被传播到最顶层，那么系统将调用一个默认的异常处理块。

3) 操作数栈区

机器指令只从操作数栈中获取操作数，对它们进行操作，并把结果返回到栈中。选择栈结构的原因是：在只有少量寄存器或非通用寄存器的机器(如 Intel486)上，也能够高效地模拟虚拟机的行为。操作数栈是 32 位的，它用于给方法传递参数，并从方法接收结果，也用于支持操作的参数，并保存操作的结果。例如，iadd 指令将两个整数相加，相加的两个整数应该是操作数栈顶的两个字。这两个字是由先前的指令压进堆栈的。这两个整数将从堆栈弹出、相加，并把结果压回到操作数栈中。

每个原始数据类型都有专门的指令对它们进行必需的操作。每个操作数在栈中需要一个存储位置，除了 long 和 double 型，它们需要两个位置。操作数只能被适用于其类型的操作符所操作。例如，压入两个 int 类型的数，如果把它们当作是一个 long 类型的数则是非法的。在 Sun 的虚拟机实现中，这个限制由字节码验证器强制实行。但是有少数操作(操作符 dupe 和 swap)用于对运行时数据区进行操作时是不考虑类型的。

4. 无用单元收集堆

Java 的堆是一个运行时数据区，类的实例(对象)从中分配空间。Java 语言具有无用单元收集能力，它不给程序员显示释放对象的能力。Java 不规定具体使用的无用单元收集算法，可以根据系统的需求使用各种各样的算法。

5. 方法区

方法区与传统语言中的编译后代码或是 UNIX 进程中的正文段类似。它保存方法代码(编译后的 Java 代码)和符号表。在当前的 Java 实现中，方法代码不包括在无用单元收集堆中，但计划在将来的版本中实现。每个类文件包含了一个 Java 类或一个 Java 界面编译后的代码。可以说类文件是 Java 语言的执行代码文件。为了保证类文件的平台无关性，Java 虚拟机规范中对类文件的格式也做了详细的说明。其具体细节请参考 Sun 公司的 Java 虚拟机规范。

1.5.5 探索 Java 虚拟机家族成员的发展史

1996 年，Sun 发布了一个虚拟机版本——JDK 1.0，其中所包含的 Sun Classic VM，不知不觉间已经历了 16 个年头，在这 16 年间涌现出许多杰出的虚拟机作品，也湮灭过许多一时大受欢迎的虚拟机作品。下面将和读者朋友们一起回顾一下 Java 虚拟机家族成员的发展历程。

1) 原始鼻祖——Sun Classic/Exact VM

1996 年 1 月，Sun 发布了 JDK 1.0，JDK 1.0 中的虚拟机就是 Classic VM。这款虚拟机

只能使用纯解释器方式来执行 Java 代码，如果要使用 JIT 编译器那就必须进行外挂，但是假如外挂了 JIT 编译器，JIT 编译器就完全接管了虚拟机的执行系统，解释器便不再工作了。如果在这款虚拟机上执行如下命令：

```
java -version
```

将会看到类似下面的输出：

```
java version "1.2.2"
Classic VM (build JDK-1.2.2-001, green threads, sunwjit)
```

其中“sunwjit”是 Sun 提供的外挂编译器，其他类似的外挂编译器还有 Symantec JIT 和 shu JIT 等。由于解释器和编译器不能配合工作，这就意味着如果要使用编译器执行，编译器就不得不对每一个方法，每一行代码都进行编译，而不论它们执行的频率是否具有编译的价值。基于程序响应时间的压力，这些编译器根本不敢应用编译耗时稍高的优化技术，因此这个阶段的虚拟机即使用了 JIT 编译器输出本地代码，执行效率和 C/C++程序也有很大差距。所以，Java 语言被扣上了一个“很慢”的帽子。

为了提升 Java 的运行效率，在 Solaris 平台上为 JDK 1.2 发布了一款名为 Exact VM 的虚拟机。Exact VM 因使用准确式内存管理而得名，即虚拟机可以知道内存中某个位置的数据具体是什么类型。假如内存中有一个 32bit 的整数 123456，它到底是一个 reference 类型指向 123456 的内存地址，还是一个数值为 123456 的整数，虚拟机有能力分辨出来，这样才能在 GC 的时候准确判断堆上的数据是否还可能被使用。由于使用了准确式内存管理，Exact VM 可以抛弃掉以前 Classic VM 基于 handler 的对象查找方式，这样每次定位对象都少了一次间接查找的开销，提升了执行性能。

注意：Exact VM 抛弃 Classic VM 基于 handler 对象查找方式的原因。

原因是 GC 后对象将可能会被移动位置，如果地址为 123456 的对象移动到 654321，在没有明确信息表明内存中哪些数据是 reference 的前提下，那虚拟机是不敢把内存中所有为 123456 的值改成 654321 的，所以要使用句柄来保持 reference 值的稳定。

实话实说，Exact VM 确实比 Classic VM 快，但是在商业应用上很快就被更为优秀的 HotSpot VM 所取代。与之形成鲜明对比的是，Classic VM 的生命周期反而更长，在 JDK 1.2 之前，Classic VM 是 Sun JDK 中唯一的虚拟机。并且在 JDK 1.2 时，Classic VM 与 HotSpot VM 并存，但默认是使用 Classic VM(可以使用“java -hotspot”参数切换至 HotSpot VM)。在 JDK 1.3 中，HotSpot VM 是默认虚拟机，它仍作为虚拟机的“备用选择”发布(可以使用“java -classic”参数切换)。从 JDK 1.4 开始，Classic VM 才正式退出商用虚拟机的历史舞台，与 Exact VM 一起逐渐淡出了我们的视线，默默无闻地进入了 Sun Labs Research VM 之中。

2) 帝国斜阳——HotSpot VM

HotSpot VM 是 Sun JDK 和 OpenJDK 中所带的虚拟机，因为是目前使用范围最广的 Java 虚拟机，所以被广大 Java 程序员所熟知。HotSpot VM 最初是由 Longview Technologies 公司设计的，而且在设计之初并非是为 Java 语言而开发的。Sun 在 1997 年收购了 Longview Technologies 公司，从而获得了 HotSpot VM。

HotSpot VM 既继承了 Sun 之前两款商用虚拟机的优点，例如准确式内存管理的优

点。除此之外，也具有许多自己新的技术优势，例如名称中的 HotSpot 指的就是它的热点代码探测技术。HotSpot VM 的热点代码探测能力可以通过执行计数器找出最具有编译价值的代码，然后通知 JIT 编译器以方法为单位进行编译。如果一个方法被频繁调用，或方法中回边(是指程序向后跳转的行为)次数很多，将会分别触发标准编译和 OSR(栈上替换)编译动作。通过编译器与解释器恰当地协同工作，可以在最优化的程序响应时间与最佳执行性能中取得平衡，而且无需等待本地代码输出才能执行程序，即时编译的时间压力也相对减小，这样有助于引入更多的代码优化技术，输出质量更高的本地代码。

2006 年的 JavaOne 大会上，Sun 宣布最终会把 Java 开源，并在随后的一年中，陆续地将 JDK 的各个部分(其中当然也包括了 HotSpot VM)在 GPL 协议下公开了源码，并在此基础上建立了 OpenJDK。这样，HotSpot VM 便成了 Sun JDK 和 OpenJDK 两个实现极度接近的 JDK 项目的共同虚拟机。

2008 年和 2010 年，Oracle 分别收购了 BEA 和 Sun 公司，这样 Oracle 就同时拥有了最优秀的两款 Java 虚拟机：JRockit VM 和 HotSpot VM。Oracle 宣布在不久的将来(大约应在 JDK 8 的时候)会完成这两款虚拟机的整合工作，使之优势互补。整合的方式大致上是在 HotSpot 的基础上，移植 JRockit 的优秀特性，譬如使用 JRockit 的垃圾回收器与 MissionControl 服务，使用 HotSpot 的 JIT 编译器与混合的运行时系统。

3) 为特定应用推出的产品——Mobile-Embedded VM / Meta-Circular VM

除了面向服务器和桌面领域的商用虚拟机外，Sun 公司还为移动和嵌入式市场提供了虚拟机产品，并且为研究人员和技术人员推出了专门的虚拟机产品，这些虚拟机对于大部分不从事相关领域开发的 Java 程序员来说可能比较陌生。Sun 公司为特定应用推出的 Java 虚拟机产品有如下几种。

(1) KVM

KVM 中的 K 是 Kilobyte 的意思，强调简单、轻量和高度可移植性，但是运行速度比较慢。在 Androd、iOS 等智能手机操作系统出现前曾经在手机平台上得到了非常广泛的应用。

(2) CDC/CLDC HotSpot

CDC/CLDC 全称是 Connected(Limited)Device Configuration。在 JSR-139/JSR-218 规范中进行定义，CDC/CLDC 希望在手机、电子书、PDA 等设备上建立统一的 Java 编程接口，而 CDC HotSpot VM 和 CLDC HotSpot VM 则是它们的一组参考实现。CDC/CLDC 是整个 Java ME 的重要支柱，但从目前 Android 和 Apple iOS 二分天下的移动数字设备市场看来，在这个领域中，Sun 的虚拟机所面临的局面远不如服务器和桌面领域乐观。

(3) Squawk VM

Squawk VM 是由 Sun 开发的、运行于 Sun SPOT(一种手持的 Wifi 设备)的虚拟机，也曾经运用于 Java Card。这是一个 Java 代码比重很高的嵌入式虚拟机实现，其中诸如类加载器、字节码验证器、垃圾收集器、解释器、编译器和线程调度都是 Java 语言本身所完成的，仅仅靠 C 语言来编写设备 I/O 和必要的本地代码。

(4) JavaInJava

JavaInJava 是 Sun 公司 1997—1998 年间所研发的一个实验室性质的虚拟机，从名字就

可以看出，它试图以 Java 语言来实现 Java 语言本身的运行环境，既所谓的“元循环”(Meta-Circular，是指使用语言自身来实现其运行环境)。它必须运行在另外一个宿主虚拟机之上，内部没有 JIT 编译器，代码只能以解释模式执行。在上世纪末主流 Java 虚拟机都未能很好解决性能问题的时代，开发这种项目，其执行速度大家可想而知。

(5) Maxine VM

Maxine VM 和上面的 JavaInJava 非常相似，它也是一个几乎全部以 Java 代码实现(只有用于启动 JVM 的加载器使用 C 语言编写)的元循环 Java 虚拟机。这个项目于 2005 年开始，到现在仍然在发展之中，比起 JavaInJava，Maxine VM 就显得“靠谱”很多，它有先进的 JIT 编译器和垃圾收集器(但没有解释器)，可在宿主模式或独立模式下执行，其执行效率已经接近了 HotSpot Client VM 的水平。

4) 其他厂家的产品——BEA JRockit / IBM J9 VM

除了 Sun 公司开发虚拟机产品外，还有很多知名的组织和公司也开发出很多优秀的虚拟机产品，其中最著名的是 BEA 公司的 JRockit 和 IBM 公司的 J9 VM。

(1) JRockit VM

JRockit VM 是 BEA 公司在 2002 年从 Appeal Virtual Machines 公司收购获得的虚拟机，曾经号称是“世界上速度最快的 Java 虚拟机”。BEA 将其发展为一款专门为服务器硬件和服务端应用场景高度优化的虚拟机，由于专注于服务端应用，它可以不太关注于程序启动速度，因此 JRockit 内部不包含解析器实现，全部代码都靠即时编译器编译后执行。除此之外，JRockit 的垃圾收集器和 MissionControl 服务套件等部分的实现，在众多 Java 虚拟机中也一直处于领先水平。

(2) J9 VM

J9 VM 并不是 IBM 公司唯一的 Java 虚拟机，只是目前 IBM 全力发展并推广的 Java 虚拟机。J9 只是内部开发代号，正式名称是 IBM Technology for Java Virtual Machine，简称 IT4J。因为这个名字太拗口，普及程度不如 J9，所以一直被称为 J9。J9 VM 最初是由 IBM Ottawa 实验室一个 SmallTalk 的虚拟机扩展而来的，当时这个虚拟机有一个 bug 是因为 8k 值定义错误引起，工程师们花了很长时间终于发现并解决了这个错误，此后这个版本的虚拟机就被称为 K8 了，后来扩展出支持 Java 的虚拟机就被称为 J9 了。与 BEA JRockit 专注于服务端应用不同，IBM J9 的市场定位与 Sun HotSpot 比较接近，它是一款设计上从服务端到桌面应用再到嵌入式都全面考虑的多用途虚拟机，J9 的开发目的是作为 IBM 公司各种 Java 产品的执行平台，它的主要市场在和 IBM 产品(如 IBM WebSphere 等)搭配以及在 IBM AIX 和 z/OS 这些平台上部署 Java 应用。

1.6　Java 的最大优势——平台无关性

Java 体系结构通过很多方面提供对平台无关性的支持。其中 Java 平台是最主要的方面。Java 平台扮演了一个运行时 Java 程序与其下操作系统和硬件之间的缓冲角色。换个角度看，Java 平台实际上充当了一个标准的 OS，只是这个 OS 目前看起来还过于笨重。Java

编程语言本身对平台无关性也提供了相应的支持，但是这是建立在 Java 平台的基础上的。对于 C/C++程序员来说，最明显的一个例子就是对基本数据类型的值域和行为的定义。以前我们一直强调不同的平台，不同的机器，不同的 CPU 都有不同的字节长度定义，但是在 Java 中，已经不存在这种差别了，因为 Java 语言统一了这些。至于究竟实际的硬件是怎么工作的，对于 Java 程序员来说，已经不用去关心了。Java 平台已经自动地完成了这种转换。

另外一个对平台无关性提供支持的方面是 Java 的 Class 文件格式。它对于所有的 Java 虚拟机都是标准统一的，这也为跨平台传递程序代码提供了实现的可能。所以，总体上来看，Java 体系结构对平台无关性的支持，主要来自于 Java 平台的设计，相当于为不同的硬件和操作系统提供了一层标准的外壳，Java 体系结构将操作建立在一个虚拟的平台上，也就达到了平台无关的目的，虽然同时也带来了很多的缺点。但是这种趋势是不可阻挡的。

说到平台无关性，不可忽略的是 Java 不仅在计算机领域实现了跨平台，而且在智能设备领域也实现了跨平台。这是通过定义不同的虚拟机标准来实现的。也就是我们耳熟能详的 J2EE、J2SE、J2ME。实际上，这三个 Java 版本就是在不同的智能设备上的虚拟机规范。而在 J2ME 领域，平台无关性体现得就更明显了，因为移动/嵌入式设备的厂商和标准都比计算机领域多得多，统一的难度就更大了。所以在 J2ME 领域，除了有 J2ME 这个大框架，还针对各个更细小的领域定义了各自的虚拟机标准，这些标准被称作 profile。

虽然平台无关性看起来是那么的诱人，但是在编写 Java 程序的时候，平台无关性只是一个选项，并不是一个必选项。从根本上来说，任何 Java 程序的平台无关程度都依赖于作者怎么编写它。有如下 6 个因素会影响平台无关性。

1) Java 平台的部署

Java 平台的平台无关性决定了要运行 Java 程序就必须要在客户机上安装符合客户操作系统的 Java 平台，也就是 Java 虚拟机。这并不是每一台计算机每一种操作系统都能满足的条件。但是幸运的是 Java 已经得到了广泛的推广，所以这个问题在大部分客户那里已经得到了解决。

2) Java 平台的版本

这个问题主要集中在 Java 不同版本的 Java API 上。虽然 Java 平台比较稳定，但是 Java API 的改动是比较频繁的，这种改动会导致新版本的 Java 程序不能在旧平台上运行，这也是开发语言的通病。

3) 本地方法

当 Java 要用到一些操作系统平台的特性，需要使用本地方法的时候，就会破坏平台无关性。Sun 一直在大力推广纯 Java 程序，就是想通过程序员的努力，尽量避免使用本地方法。

4) 非标准运行时库

虽然本地方法会破坏平台无关性，但是 Java API 提供的标准方法中使用的本地方法却不在其列。在标准 API 之外，还有一些厂商和组织提供的非标准方法，这些方法有可能使用了本地方法，却不是任何 Java 平台都拥有的，所以对于这种非标准方法，一定要谨慎使用，如果它们使用了本地方法来实现，那么平台无关性就被破坏了。

5) 对虚拟机的依赖

在分析 Java 虚拟机的时候，需要牢记的一点是，虚拟机不过是 Sun 提供的一个文字标准，并不是具体的实现，我们平时用的虚拟机是 Sun 提供的，但是这并不是标准的虚拟机实现，实际上虚拟机并没有标准的实现，唯一的标准就是 Sun 给出的那份标准。所以对于虚拟机标准提供的一些特性，千万不能作为程序逻辑的一部分。因为标准只是指出一定要实现这些特性，但是具体怎么实现，是由具体的虚拟机实现决定的。一些可以举例的特性是垃圾回收机制、线程优先级等。它们被实现了，但是并没有一个确定的实现方式，比如垃圾回收的时机，线程优先级高低是否一定决定了 CPU 时间分配的多少等。

6) 对用户界面的依赖

在不同的 Java 平台上，要保证每个平台的 UI 都让用户满意是很困难的事情。所以在跨平台设计的时候，要谨慎地设计 UI 的接口，以保证在每个平台上都有令人满意的表现。

1.6.1　平台无关性的好处

Java 平台无关性的最大好处是，使用开发的软件产品，不需改动代码或只做很小的更改就能在任何主流的平台上运行，给公司节约很大的开发成本。无论是在网络应用方面，在嵌入式应用方面，还是对开发人员来说，Java 的平台无关性都给用户带来了巨大的好处。

1．网络应用

Java 技术在网络环境下非常有用，这是因为用 Java 创建的可执行二进制代码能够不加改变地运行于多个平台。这一点在网络化环境中特别重要，因为大多数网络通常都是由各种各样不同种类的计算机和设备互联而成。例如，网络上可能链接了艺术创作部门的 Macintosh 计算机、工程部门的 UNIX 工作站以及随处可见的运行 Windows 的 PC。尽管这种情形下，公司内部的各种计算机和设备可以共享彼此的数据，但是它仍然需要大量的管理工作。像这样一个网络，要求系统管理员必须随时维持运行于不同种类计算机上的同一个程序，在更新的时候要根据特定于它所运行的不同平台进行版本同步更新。如果程序能够不加修改地运行于网络上的任何计算机，而不管该计算机是什么种类，那么这将极大地减轻系统管理员的工作。特别是当这样的程序是通过网络交付的时候，效果更加显著。

2．嵌入式应用

很多网络化嵌入式设备展示了 Java 另一个擅长的领域。例如，在工作场所中的打印机、扫描仪和传真机等嵌入式设备，它们通常都连接到了内部网络中，像这样连接到网络的嵌入式设备，也可以出现在消费品领域，像家庭网络和汽车等。在嵌入式的世界中，Java 的平台无关性也有助于简化系统管理任务。通过专用于给网络带来即插即用功能的技术，就极大地减少了在网络互联的嵌入式设备环境下的管理任务，不管是对在家里的消费者还是在工作场所的系统管理员都一样。一旦某个设备加入到这样的网络中，它就能立即访问网络上的其他设备，同样其他的设备也可以访问它。为了达到如此简单易用的连接能力，采用了 Jini 技术的设备将通过网络彼此交换对象，要是没有 Java 对平台无关性的支

持，这绝对不可能做到。

3．开发人员

对于开发人员来说，Java 能够减少开发和在多个平台上部署应用程序的成本和时间。如果想要支持多个平台，相对于得到的回报而言会显得成本过于高昂，所以当今大多数程序都只能支持一个平台。但是因为 Java 能减少支持多个平台所花的代价，所以这些代价对于很多程序来说是合算的。

对于软件开发者来说，Java 的平台无关性既有有利的一面，也有不利的一面。

- 有利的一面：如果正在开发和销售的某个软件产品，Java 的平台无关性有助于商家进入到更多的市场，而不是开发一个仅能运行于 Windows 的程序。例如，可以开发一个能同时运行于 Windows、OS/2、Solaris 和 Linux 的程序。使用 Java 可以帮助商家拥有更多的潜在客户。
- 不利的一面：别人同样也能这么做。假如你正在集中精力为 Solaris 开发一个非常棒的软件，Java 却让其他人更容易开发出类似的软件，并和你在同一个目标市场中竞争，在使用 Java 时，不能只看到它带来更多潜在客户的好处，同样它也带来更多潜在的竞争对手。

在程序员眼中，Java 程序具备了不加修改便可以运行于多个平台的能力，这给予了网络一个同构的运行环境。这就使得新的分布系统可以围绕着“网络移动”对象来构建。

1.6.2 Java 对平台无关性的支持

Java 为了支持 Java 程序的平台无关性，通过 Java 平台、Java 语言、Java Class 文件和可伸缩性等 4 个方面进行了支持。

1．Java 平台

Java 平台扮演了一个运行时 Java 程序与其下的硬件和操作系统之间的缓冲角色。Java 程序被编译为可运行在 Java 虚拟机中的二进制程序，并且假定 JavaAPI 的 Class 文件在运行时都是可用的。接下来虚拟机运行程序，那些 API 则给予程序访问底层计算机资源的能力。无论 Java 程序被部署到何处，它只需要与 Java 平台交互，而不需要担心底层的硬件和操作系统，所以它就能够运行于任何拥有 Java 平台的计算机。

2．Java 语言的支持

Java 语言的基本数据类型的值域和行为都是由语言自己定义的。而在 C 语言或 C++语言中，基本整数类型 int 的值域是由它的占位宽度决定的，而它的占位宽度则由目标平台决定。一般来说，C 或 C++中 int 的占位宽度是由编译器根据目标平台的字长来决定。这就意味着针对不同平台编译的同一个 C++程序在运行时可能会有不同的行为，这仅仅是因为基本数据类型在不同的平台上值域不同。但是对于 Java 程序来说，不管其运行的平台是什么，Java 中的 int 都是 32 位二进制补码表示的有符号整数，而 float 则总是遵守 IEEE754 浮点标准的 32 位浮点数。并且这一点在 Java 虚拟机内部以及 Class 文件中都是一致的。通过确保基本数据类型在所有平台上的一致性，Java 语言本身为 Java 程序的平台无关性提

供了强有力的支持。

3．Java Class 文件的支持

Java 的 Class 文件定义了一个特定于 Java 虚拟机的二进制格式，此文件可以在任何平台上创建，也可以被任何平台的 Java 虚拟机装入并运行。Class 文件的格式都有严格的定义，例如，多字节值的高位优先存放约定，并且与 Java 虚拟机所在平台无关。

4．可伸缩性

Java 支持平台无关性，主要体现在它的可伸缩性。Java 平台可以在各种不同类型的计算机上实现，无论是嵌入式设备还是大型主机。

虽然 Java 目前在 Web 领域和桌面领域都声明卓著，但它最初确实是被期望用于嵌入式设备和消费电器领域的，而不是桌面计算机。这样的设计目标，部分原因是由于，尽管 Microsoft 公司和 Intel 公司在桌面市场占有统治地位，但是它们在嵌入式设备和消费电器市场并不具备这种优势。微处理器开始越来越多地出现在音频视频装备、蜂窝电话、打印机、传真机、复印机等设备中，而这些微处理器可以连接到网络。因而，Java 最初的设计目标之一就是提供某种方式，让软件可以通过网络交付到任意种类的嵌入式设备中——不管它的微处理器和操作系统是什么。

为了达到这个目标，Java 运行时系统(Java 平台)不得不设计得尽量紧凑，以便它可以使用嵌入式系统中有限的资源以软件的方式来实现，嵌入式微处理器通常有一些特殊的限制，比如，很少的内存、没有磁盘、没有图形化显示，甚至根本没有显示功能。这样的限制，也就意味着嵌入式设备和消费品系统通常没有必要，或者不可能支持所有的 JavaAPI。

针对嵌入式和消费性电器设备的特殊需求，Sun 创建了几个具体的 Java 平台，他们包含更少的 API。

- Java 个人版平台，用于消费性电器设备。
- Java 嵌入式平台，用于嵌入式系统。
- Java 卡平台，用于智能卡。

上述 Java 平台是由 Java 虚拟机和比标准的 Java 平台更小的运行时库组成的。由此可见，个人版平台和标准版平台的区别是：前者比后者提供的 JavaAPI 运行时库内容更少，而嵌入式平台则比个人版平台还要少。虽然上述 Java 平台依次面向更小的执行环境，并且在资源利用上有更严格的限制，但是他们所提供的 API 之间并非是简单的子集关系。每一个平台提供的 API 子集都是面向一个特定的目标领域，因此，也包含专门针对该目标领域的 API。

因为 Java 平台很紧凑，所以可以在很多嵌入式系统和消费性电器中实现。Java 平台蕴含的紧凑性并没有把实现限制在很小的范围。Java 平台可以在个人计算机、工作站和大型机上保持伸缩性。虽然在 Java 开始的几年里，Java 虚拟机在服务器端遇到过伸缩性的困难，但是虚拟机现在已经针对服务器做过优化，很多实现可以在服务器端得到非常好的性能。在这个方面，Sun 定义了一个 API 超集——J2EE，后来升级为 JavaEE。除了标准的 JavaAPI 之外，JavaEE 还包含了在企业服务环境中非常有用的一些技术，例如，Servlet 和

EnterpriseJavabean。

最终 Sun 改变过的 API 定义方式保留了三个基础 API 集合，他们实现了 Java 平台不同的伸缩性：

- 企业版 J2EE，后来升级为 JavaEE。
- 标准版 J2SE。
- 微型版 J2ME。

在高端应用领域，企业版的存在表明了 Java 平台在高端服务的可用性。在中端应用领域，标准版提供了在浏览器中启动传统 applet 的功能和桌面环境下的 Java 平台。在低端应用领域，微型版通过不同的行业子集，显示了 Java 平台可以向下伸缩，并改变自己以适应完全不同的消费性电器市场和嵌入式系统需求的能力。

进入 2012 年之后，Java 在嵌入式的地位稳步提升，例如，很火的 Android 应用开发，默认的语言就是 Java。

1.6.3 分析影响 Java 平台无关性的因素

Java 的体系结构不仅有利于创建平台无关性软件，而且可以快速地创建和平台相关的软件。当编写 Java 应用程序时，平台无关性只是一个可选的性能。影响 Java 程序平台无关性的因素有很多，主要有 Java 平台的部署、Java 平台的版本、本地方法、非标准运行时库、对虚拟机的依赖、对用户界面的依赖、Java 平台实现中的 bug 和测试等。在上述因素中，其中有一些因素不在开发人员的控制范围之内，但是大多数是由开发人员来控制的。从根本上说，任何 Java 程序的平台无关程度都依赖于作者怎样编写它。

1. Java 平台部署因素

Java 平台在不同的平台上被部署的程度是决定 Java 程序平台无关性的主要因素。因为只有在拥有 Java 平台的计算机或设备上才能运行 Java 程序，所以要想在一台特定的计算机上运行自己编写的 Java 程序时，则必须将 Java 平台移植到拥有特定类型的硬件和操作系统之上。假设某些 Java 平台的开发商已经完成了移植的实现，那么这个实现接口还必须用某种方法安装到我们的计算机上。因此决定 Java 程序平台无关性真正程度的一个重要因素——这个因素一般不是由开发人员控制的——就是已有了可用的 Java 平台实现和发布版本。

对于 Java 开发人员来说，Java 平台的部署因其广泛的需求性已得到推广。无论是 Web 浏览器，还是桌面计算机、工作站、网络操作系统和嵌入式设备，它们都需要 Java 平台。所以我们非常有可能已经在自己的计算机或设备上拥有了 Java 平台的实现。

2. Java 平台版本因素

Java 平台中保证可用的基本库集合被称为标准 API。Sun 把 Java 虚拟机以及组成标准 API 的那些 Class 文件成为 Java 平台标准版。这个版本的 Java 平台是 JavaAPI 库的最小集合，例如，可以在普通桌面电脑和工作站上使用。Sun 同时也定义了 Java2 平台的微型版和企业版，并鼓励在各种消费电器和嵌入式设备行业开发 API 子集以加强微型版。除此之外，Sun 还定义了一些标准运行时库，它把这些库作为标准版的可选项，把它们策划成为

标准扩展 API。这些库包括一些如电话、商业性的服务，并且还包括诸如音频、视频或 3D 类的多媒体服务。如果在程序中使用了标准扩展 API 中的库，它可以在任何支持标准扩展 API 的地方运行，但是不能在一个只安装了最基本的标准平台的计算机上运行。除此之外，一些标准扩展 API 在企业版的任何实现中都保证可用，有了这个 API 的版本以及协议，Java2 平台就不能只代表一个同构的执行环境了——同构的执行环境可以使一段代码一次编写多处运行。

在某种意义上，Java 平台是随着时间而不断发展的。虽然 Java 虚拟机发展得非常缓慢，但是 Java API 会非常频繁地被改动。随着时间的推移，不论是标准版还是标准扩展 API，都将加入或删减某些特性，甚至标准扩展 API 中的一部分可能会移植到标准版中。在改动 Java 平台时，大部分都必须是向上兼容的，这就意味着它们不能破坏已有的 Java 程序，但是有些改动也未必一定要这样。例如：有些过时的特性已在新版本的 Java 平台中删除，一些已存在的特性的程序将不能在新版本中运行；而且改动也可能不向下兼容，这样针对 Java 平台新版本而编写的程序就不一定能在老版本中运行。

Java 平台的动态特性在一定程度上使事情变得复杂，因为开发人员希望能够只写一个程序就可以在任何计算机上运行。在理论上，只要程序仅仅依赖于标准 API 的运行时库，那么程序就应该可以在有 Java2 平台标准版的所有计算机上运行。但实际上，标准 API 的新版本要过一段时间才能在任何地方都适用。当程序依赖于标准 API 最新版本的一些特性时，有些主机可能不能运行这个程序，因为他们只有比较老的版本，这对于软件开发人员来说已不是一个新问题，例如，为 Windows 7 编写的程序，不能在以前的操作系统 Windows XP 中工作，但是因为 Java 实现了软件的网络分发，这个问题就更加尖锐了。Java 的好处不仅在于它可以方便地使程序从一个平台移植到另一个平台，而且可以使同一段二进制 Java 代码通过网络发送，并且可以在任何计算机或设备上运行。

作为开发人员来说，虽然我们不能控制 Java 平台版本的发布周期或者部署进度，但是可以为自己的程序选择所依赖的 Java 平台版本。当官方发布一个新的 Java 平台版本时，我们必须自行确定基于哪一个版本来编写程序代码。

3. 本地方法

是否调用了本地方法是决定 Java 程序的平台无关程度的一个主要因素。当编写一个平台独立的 Java 程序时，必须遵守的一条最重要的原则就是：不要直接或间接调用不属于 Java API 的本地方法，调用 Java API 以外的本地方法将使程序平台相关。

在不需要平台无关性的情况下，可以直接调用本地方法。通常在以下三种情况下适用本地方法。

- 为了使用底层的主机平台的某个特性，而这个特性不能通过 Java API 访问。
- 为了访问一个老的系统或者使用一个已有的库，而这个系统或库不是用 Java 编写的。
- 为了加快程序的性能，而将一段时间敏感的代码作为本地方法实现。

如果必须使用本地方法，而且要使程序可以在多种平台上运行，那么必须将本地方法移植到所有需要的平台上。这种移植必须用传统的方法实现，而一旦完成了移植，必须说明怎样将这个平台相关的本地方法库分发到合适的主机。因为 Java 体系结构的设计目的是

简化多平台的支持，所以，编写平台独立的 Java 程序的最初目的是要完全禁止本地方法，并且应该仅通过 Java API 和主机交互。

4．非标准运行时库

本地方法和平台无关性不相容，平台无关性的任务是保证方法在“任何地方”都已实现，例如，Java API 可以同时在如 Windows 或 Linux 等操作系统上使用本地方法来访问主机。当调用了 Java API 中的一个方法时，可以确保它在任何地方都是可用的。而这个方法在某些地方是否是用本地方法实现的这个问题是无关紧要的。

Java 平台可以由许多开发商来实现，虽然每个开发商必须提供 Java API 的标准运行时库，但是有极个别开发商提供了另外的库。如果在开发时侧重于平台无关性，那么就必须清楚地知道所使用的那些非标准运行时库是否调用了本地方法。如果没有调用本地方法的非标准库，就不会降低程序的平台无关性。如果使用了调用本地方法的运行时库，那么就会产生和直接调用本地方法一样的结果，使得程序和平台相关了。

5．对虚拟机的依赖

在编写平台独立的 Java 程序时，还必须遵循以下两条原则。

(1) 不要依赖及时终结(Finalization)来达到程序的正确性。

(2) 不要依赖线程的优先级(Thread Prioritization)来达到程序的正确性。

上述两条原则和 Java 虚拟机中的某些部分有关，Java 虚拟机中的某些部分可以由不同的开发商用不同的方法实现。通过这两条原则，可以防止垃圾收集和线程在不同实现中的变化所带来的不利影响。

为了保证效率，所有的 Java 虚拟机都必须有垃圾收集器，但是不同的实现可能使用不同的垃圾收集技术。在 Java 虚拟机的规范中这个灵活性意味着，在不同的虚拟机中，一个特定的 Java 程序中的对象可能在不同的时间被垃圾收集。这也就意味着那些在对象被释放以前由垃圾收集器运行的终结方法，在不同的虚拟机中可能是在不同的时间运行的。如果使用了一个终结方法来释放有限的内存资源，例如，文件句柄，程序就可能可以在一些虚拟机的实现上运行，而在其他实现上却不能。在一些实现上，程序可能在垃圾收集器得到机会调用释放资源的终结方法之前，就已经将有限的资源耗尽了。

在不同 Java 虚拟机的实现中，另一个变化和线程的优先级有关。Java 虚拟机规范只保证了程序中拥有最高优先级的可运行线程会得到一些 CPU 时间。这个规范也保证了在较高优先级的线程被阻塞时，较低优先级的线程将会运行。但是，在较高优先级的线程没有被阻塞的情况下，并没有禁止较低优先级的线程的运行。在某些虚拟机的实现中，即使较高优先级的线程未被阻塞，那些较低优先级的线程也可能得到 CPU 时间。如果你的程序依赖于这个行为的正确性，它将在某些虚拟机的实现上可以正常运行，而在某些实现上却不能。为了保证多线程程序的平台独立性，必须依赖同步而不是优先级来在线程间协调相互间的动作。

6．Java 平台实现中的 bug

在 Java 平台的不同实现之间，bug 是其中的一个主要变化因素。虽然 Sun 已经开发出

了一套全面的测试标准，但是 Java 平台的实现必须通过这套测试。但是问题是，可能其中的某些实现在发布时仍然包含 bug，此时就只能通过测试来防止这种可能性。通过 bug 的存在，可以通过测试来确定这个 bug 是否影响到你的程序，如果影响则必须试图找到一个绕开的途径。

7. 测试

因为在不同 Java 平台的实现之间会存在差异，所以当依赖某些特定平台写的 Java 程序时，以及在任何特定的 Java 平台的实现中都可能有 bug 存在，所以应该尽可能在所有希望运行的平台上测试 Java 程序。就当前现实情况来说，Java 程序的平台无关性并没有达到只需要在一个平台上测试成功，就表明可以运行于其他平台的能力。我们仍然需要在其他平台上测试 Java 程序，而且应该在程序运行的主机上尽可能找到所有的 Java 平台的不同实现，在这些实现上都对程序进行测试。所以在实际情况中，在程序要运行的不同主机和不同 Java 平台实现上测试你的 Java 程序，是程序平台无关性的一个关键因素。

1.6.4 实现平台无关性的策略

Java 的体系结构允许开发人员在平台无关性和其他考虑之间进行选择。在编写程序时，通过选择所使用的方法进行选择。如果目的是使用那些平台的相关特性和一个老系统进行交互，或使用一个不是用 Java 编写的现有的库，或者得到程序的最快执行速度，那么可以使用本地方法来达到此目的。在这时程序的平台无关性将会降低，并且是可以接受的。相反，如果目的是考虑平台无关性，那么在编写程序时需要遵循一定的规则。下面列出了实现程序最佳可移植性的 7 个步骤。

(1) 选择程序要运行的主机和设备的集合，也就是“目标宿主机”。

(2) 在目标宿主机中选择自认为足够好的 Java 平台版本，在该版本 Java 平台上编写、运行程序。

(3) 为每个目标宿主机选择一些程序将要运行的 Java 平台实现。

(4) 编写程序，使其只通过 Java API 的标准运行时库来访问计算机。不要调用本地方法，或者开发商专有的那些调用本地方法的库。

(5) 编写程序，使它不依赖垃圾收集器及时终结的正确性，也不依赖线程的优先级。

(6) 努力设计一个用户界面，使它在你所有的目标宿主机上都能正常工作。

(7) 在所有的目标运行时环境和所有的目标宿主机上测试程序。

作为一个开发人员来说，当思考怎样编写某个特定的 Java 程序时，软件工业的策略和宣传不一定是主要的考虑因素。对于写的有些程序，可能适合平台无关性；而对于其他的一些，则平台相关的程序可能会更有意义一些。在每种情况下，都需要作出决定，这个决定要基于用户的需要，以及要把自己放在市场的什么位置。

如果遵从了以上列出的 7 个步骤，那么 Java 程序将肯定可以在你所有的目标宿主机上运行。如果目标宿主机涉及大多数主要的 Java 平台的开发商和大多数主要类型的计算机，那么很有可能我们的程序也能在其他地方运行。

第 2 章

JDK 编译测试

想要深入了解 JDK 内部的实现机制，需要亲自编译一套 JDK，通过阅读和跟踪调试 JDK 源码的方式可以更好地了解 Java 技术体系的原理。并且在 JDK 中的很多底层方法都是 Native(本地)的，当需要跟踪这些方法的运作或对 JDK 进行 Hack(修改)的时候，都需要自己编译一套 JDK。本章将详细讲解在不同平台下编译 JDK 的过程。

2.1 为什么要编译 JDK

对于初学者来说，要想真正编译 JDK 的话，建议尽量不要在 Windows 平台编译，因为难度比在 Linux 平台编译要高不少。哪怕读者们没有 Linux 环境，临时装一个 ubuntu，加上安装操作系统的时间都比直接在 Windows 下编译来得快。如果要在 Windows 平台编译的话，看看是否需要把整个 JDK(HotSpot、Library、Utils(如 VisualVM 等)、JAXWS、etc)都编译出来，相信大部分人只想要一个虚拟机，那可以关闭掉其他部分的编译，省不少事。所以接下来将遵循先易后难的原则，分别讲解在 Windows 平台和 Linux 平台编译 JDK 的知识。

在当前网络中有很多开源 JDK 可以供我们选择，例如 Apache Harmony 和 OpenJDK 等。因为当前 Sun 系列的 JDK 是使用得最广泛的 JDK 版本，所以本书选择使用 OpenJDK 进行编译测试。

2.2 在 Windows 平台编译 JDK

在本节的内容中，将详细地讲解在 Windows 平台编译 JDK 的基本步骤。

2.2.1 为什么选择 OpenJDK

在众多 JDK 版本中，最合适作为编译的是 OpenJDK，主要原因如下。

(1) OpenJDK 的核心代码与同时期 Sun(Oracle)的产品版基本上是一样的，血统纯正，不用担心性能问题，也基本上没什么兼容性问题。代码上最主要的差异是在原本 JDK 依赖的第三方库上，包括加密库、音频库、字体等。核心部分，也就是 HotSpot VM 与 Java 核心库基本上保持了 Sun JDK 的原貌，甚至比 Sun JDK 还更快地吸收了社区反馈的贡献。

(2) OpenJDK 是真正开源的，许可证是 GPLv2+CE，使用上比原本 JDK 的另外两种许可证要自由一些。

(3) OpenJDK 的构建系统比原本的 JDK 有大幅改进，使整个 build 过程变得非常轻松。

OpenJDK 最新的两个版本是 OpenJDK 6 和 OpenJDK 7，两者都是开源的，源码都可以在 http://openjdk.java.net/下载获得。其实 OpenJDK 6 的源码是从 OpenJDK 7 的某个基线中引出的，然后剥离掉 JDK 1.7 相关的代码，从而得到一份可以通过 TCK 6 的 JDK 1.6 实现。由此可见，直接编译 OpenJDK 7 会更加“原汁原味”一些。

2.2.2 获取 JDK 源码

有以下两种获取 OpenJDK 7 源码的方法。

(1) 使用 Mercurial 代码版本管理工具从 Repository 中直接取得源码，Repository 的地址是 http://hg.openjdk.java.net/jdk7/jdk7j。

这是一种最直接的方法，源码可以比较直观地展现在我们眼前。但是这种方法的缺点是麻烦，因为 Mercurial 远不如 SVN、ClearCase 或 CVS 之类的版本控制工具那样普及。

(2) 直接下载官方打包好的源码包，可以从 http://download.java.net/openjdk/jdk7/ 的 Source Releases 页面取得打包好的源码，如图 2-1 所示。





图 2-1　下载打包好的源码

一般来说，大概一个月左右会更新一次，虽然不够及时，但的确方便了许多。笔者下载的是 openjdk-7-fcs-src-b147-27_jun_2011.zip 版，2011 年 6 月 27 日发布的，压缩包大小大约 83.18MB。

2.2.3　系统需求

在 Windows 平台上编译 JDK 之前，请事先认真阅读源码中的 README-builds.html 文档，在此文档中阐明了我们应该注意的细节。对于初学者来说，第一次编译需要耗费很多时间是很正常的事情。

在编译时，需要将编译涉及的所有文件都存放在 NTFS 格式的文件系统中，这是因为 FAT32 格式无法支持大小写敏感的文件名。官方文档明确指出了编译所需的最低配置：512MB 的内存和 600MB 的磁盘空间。其实 600MB 的磁盘空间仅仅是指存放 OpenJDK 源码和相关依赖项的空间，要想完成编译，仅仅 600MB 的空间是不够的。这是因为在编译过程中所需要下载的工具、依赖项、源码需要超过 1GB 的空间。

另外还建议读者，不要将文件(包括源码和依赖项)放在包含中文或空格的目录里面，这样做不是一定不可以，只是这样会为后续建立 Cygwin 环境带来很多额外的工作，这是由于 Linux 和 Windows 的磁盘路径差别所导致的，我们也没有必要自己给自己找麻烦。

2.2.4 构建编译环境

在构建编译环境步骤中，需要先安装 Cygwin，这是一个在 Windows 平台下模拟 Linux 运行环境的软件，它提供了一系列的 Linux 命令支持。使用 Cygwin 的原因是，在编译过程中需要使用 GNU Make 执行 Makefile 文件。在安装 Cygwin 时不能直接默认安装，因为表 2-1 中所示的工具都不会进行默认安装，但又是编译过程中所需要的，所以需要在图 2-2 的安装界面中进行手工选择。

表 2-1 需要手工选择安装的 Cygwin 工具

文件名	分类	包	描述
ar.exe	Devel	binutils	The GNU assembler, linker and binary utilities
make.exe	Devel	make	The GNU version of the 'make' utility built for Cygwin
m4.exe	Interpreters	m4	GNU implementation of the traditional Unix macro processor
cpio.exe	Utils	cpio	A program to manage archives of files
gawk.exe	Utils	awk	Pattern-directed scanning and processing language
file.exe	Utils	file	Determines file type using ‘magic’ numbers
zip.exe	Archive	zip	Package and compress (archive) files
unzip.exe	Archive	unzip	Extract compressed files in a ZIP archive
free.exe	System	procps	Display amount of free and used memory in the system

Cygwin 安装时的定制包选择界面如图 2-2 所示。



图 2-2 Cygwin 的安装界面

1. 下载、安装 Cygwin

(1) 登录 Cygwin 的官方网站下载 Cygwin 的安装程序，其官方网站地址是 http://www.cygwin.com/。

或者直接使用下载连接来下载安装程序，下载连接是 http://www.cygwin.com/setup.exe。

(2) 下载完成后，运行 setup.exe 程序，出现安装画面。直接单击"下一步"按钮，出现安装模式对话框，如图 2-3 所示。

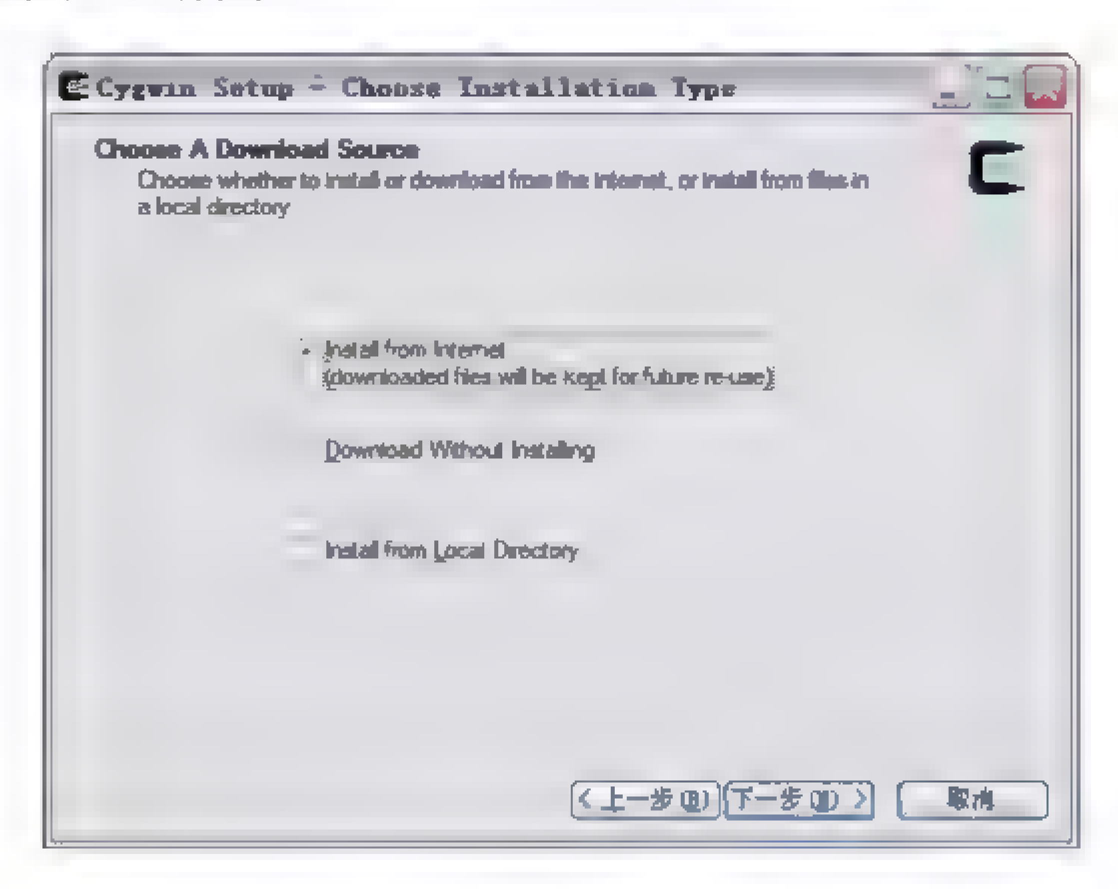


图 2-3　安装模式对话框

在上述界面中可以看到以下三种安装模式：

- Install from Internet：这种模式直接从 Internet 安装，适合网速较快的情况；
- Download Without Installing：这种模式只从网上下载 Cygwin 组件包，但不安装；
- Install from Local Directory：这种模式与第二种模式对应，当你的 Cygwin 组件包已经下载到本地，则可以使用此模式从本地安装 Cygwin。

从上述三种模式中选择适合你的安装模式，这里我们选择第一种安装模式，直接从网上安装，当然在下载的同时，Cygwin 组件也保存到了本地，以便以后能够再次安装。

(3) 选中后单击"下一步"按钮，在弹出的新界面中选择 Cygwin 的安装目录，以及一些参数的设置。默认的安装位置是"C:\Cygwin\"，你也可以选择自己的安装目录，然后单击"下一步"按钮，如图 2-4 所示。

(4) 在弹出的新界面中选择安装过程中从网上下载的 Cygwin 组件包的保存位置，如图 2-5 所示，然后单击"下一步"按钮。

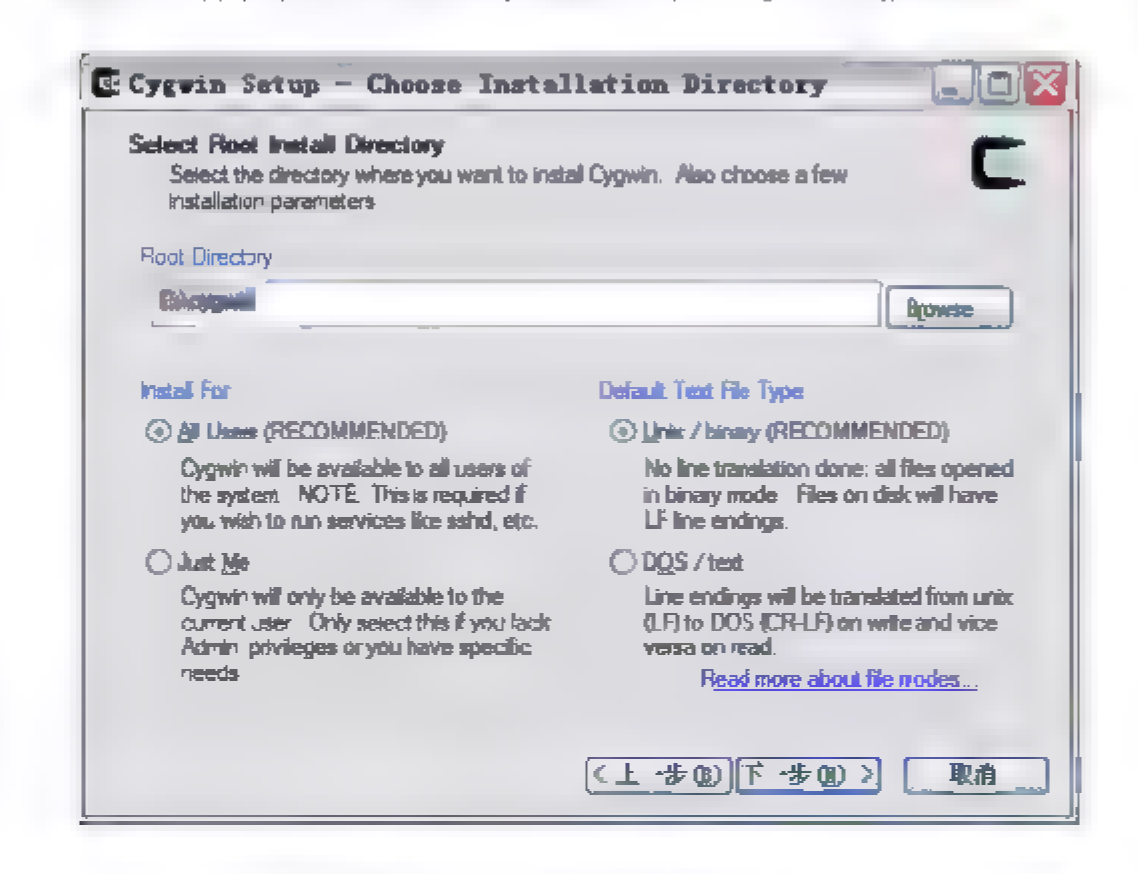


图 2-4　选择安装目录

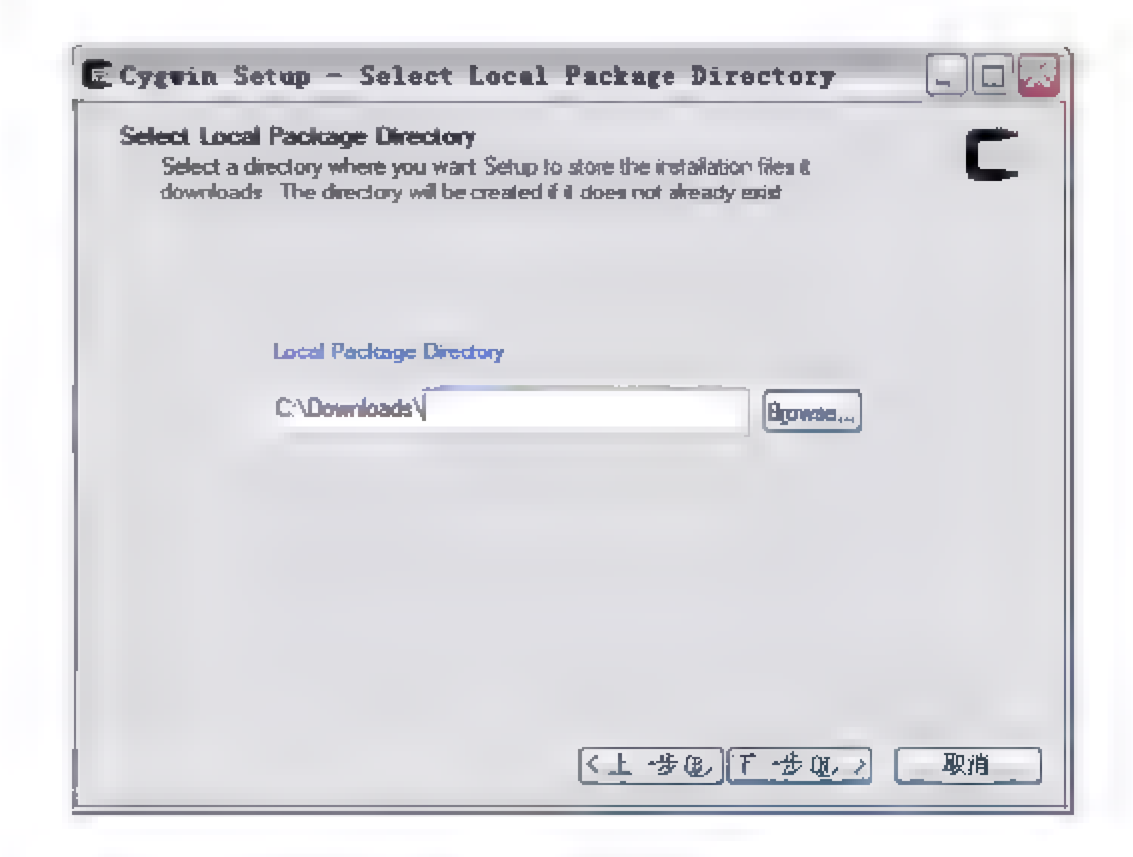


图 2-5　从网上下载的 Cygwin 组件包的保存位置

(5) 在弹出的新界面中选择连接方式，然后单击“下一步”按钮，如图 2-6 所示。

(6) 在弹出的新界面中选择下载站点，为了获得最快的下载速度，需要先在列表中寻找 Cygwin 中国镜像的地址 http://www.cygwin.cn，如果找到就选中这个地址；如果找不到这个地址，就在下面手动输入中国镜像的地址 http://www.cygwin.cn/pub/，然后再单击 Add 按钮，再在列表中选择。选择完成后，单击“下一步”按钮，如图 2-7 所示。

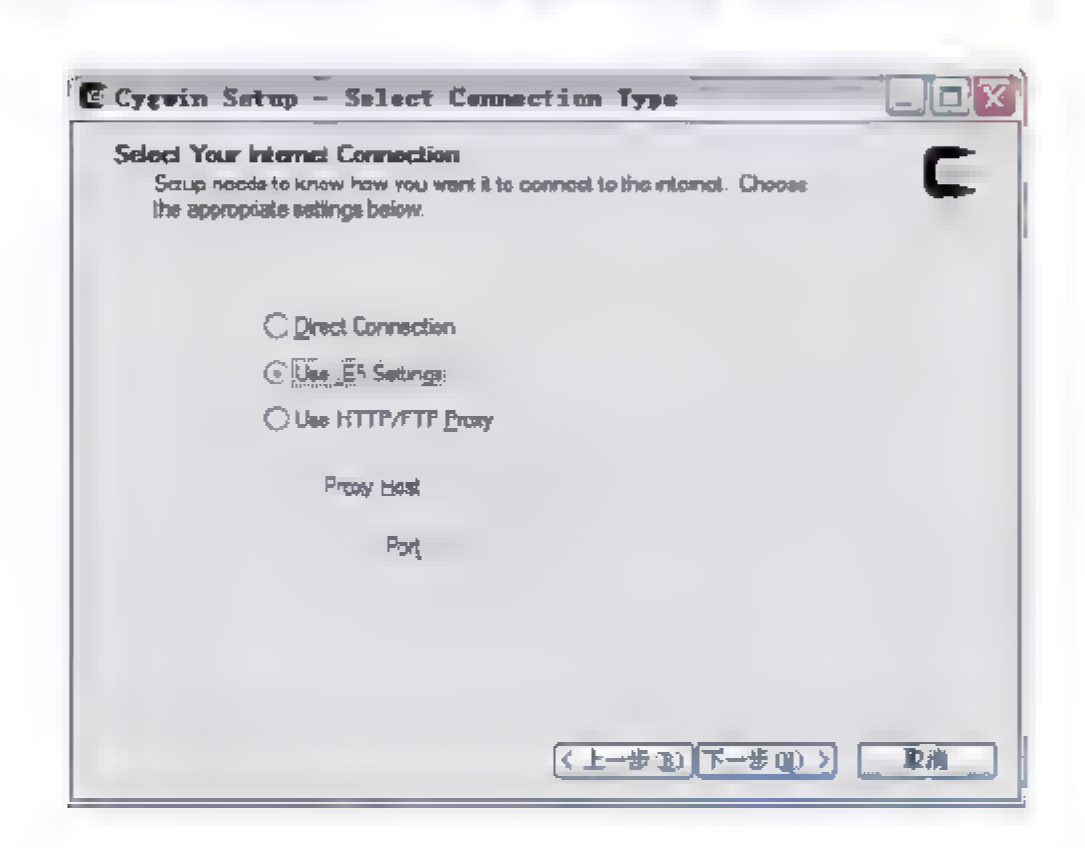


图 2-6　选择连接方式

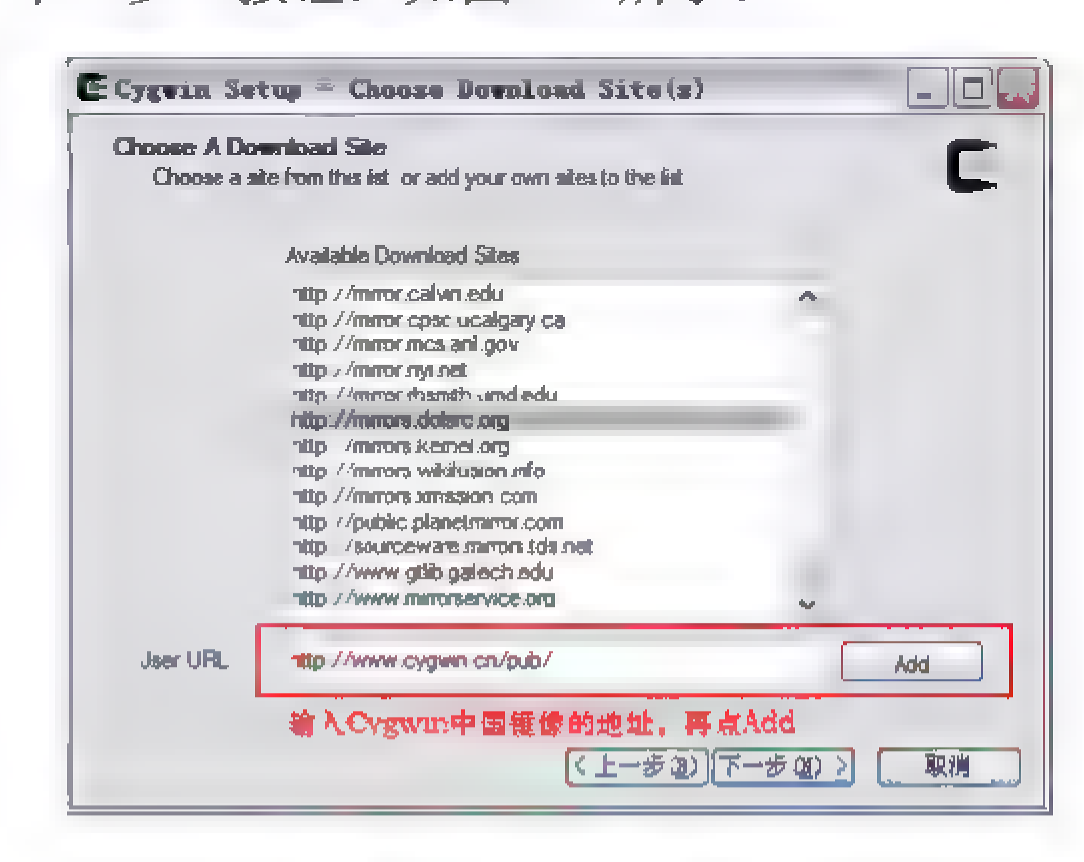


图 2-7　选择下载站点

(7) 在弹出的新界面中选择需要下载安装的组件包，为了使安装的 Cygwin 能够编译程序，需要安装 gcc 编译器。默认情况下，gcc 并不会被安装，我们需要选中它来安装。为了安装 gcc，要打开组件列表中的 Devel 分支，如图 2-8 所示。

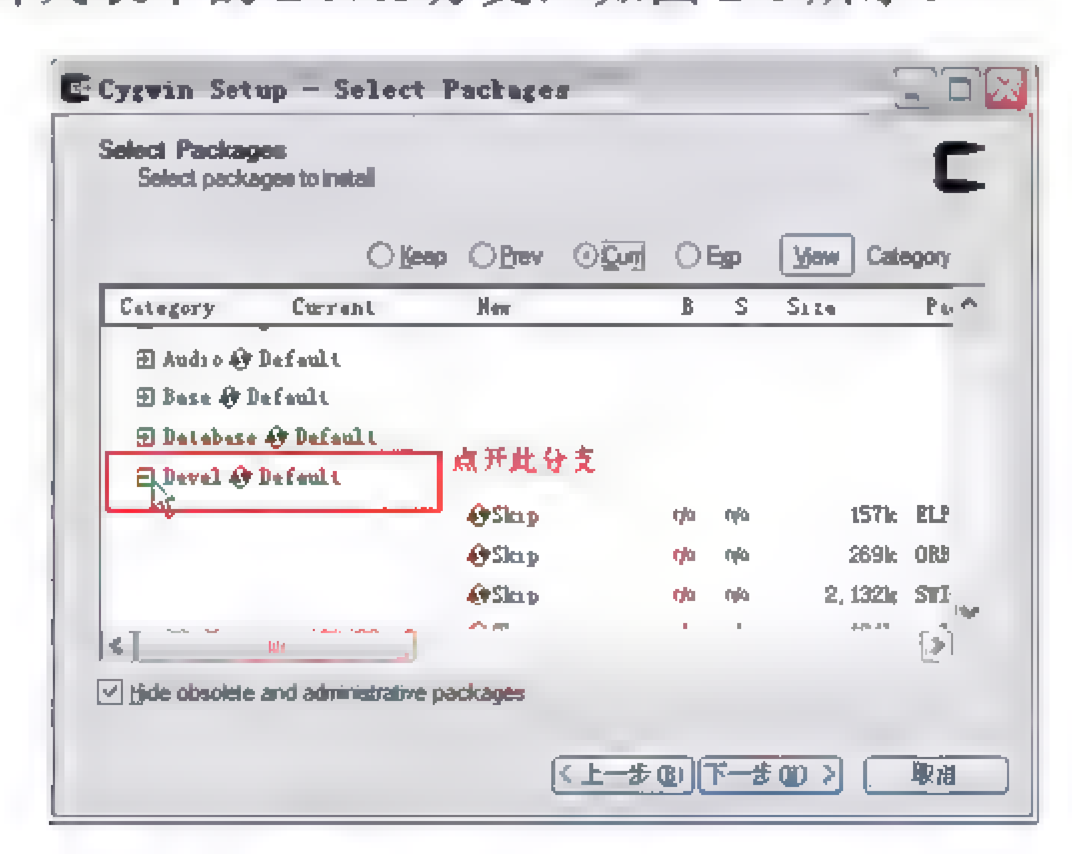


图 2-8　选择需要下载安装的组件包

在该分支下有很多组件，我们必须选择下面的选项：

- binutils
- gcc
- gcc-mingw
- gdb

单击组件前面的循环按钮，会出现组件的版本日期，在此选择最新的版本安装，图 2-9～图 2-12 是选中的四类组件的截图。

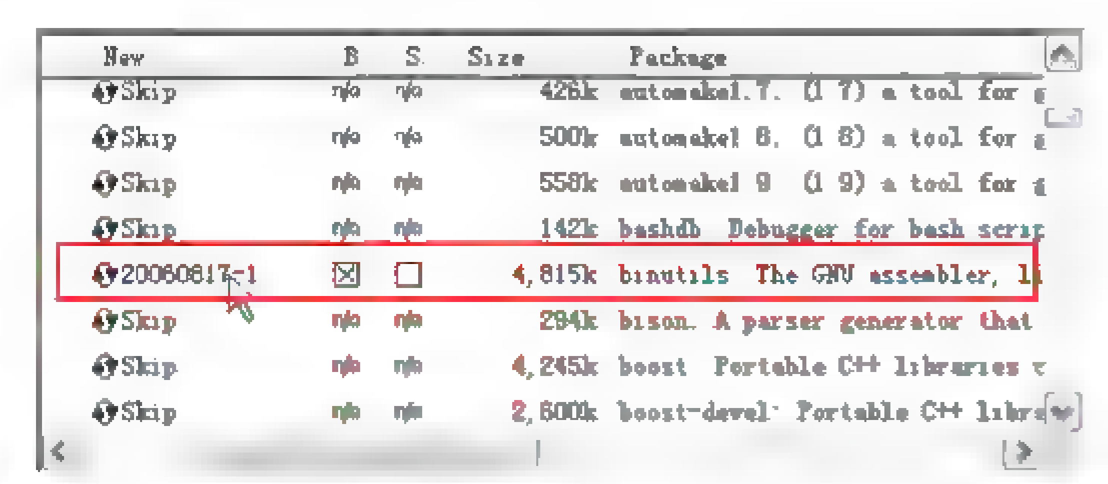


图 2-9　binutils 组件

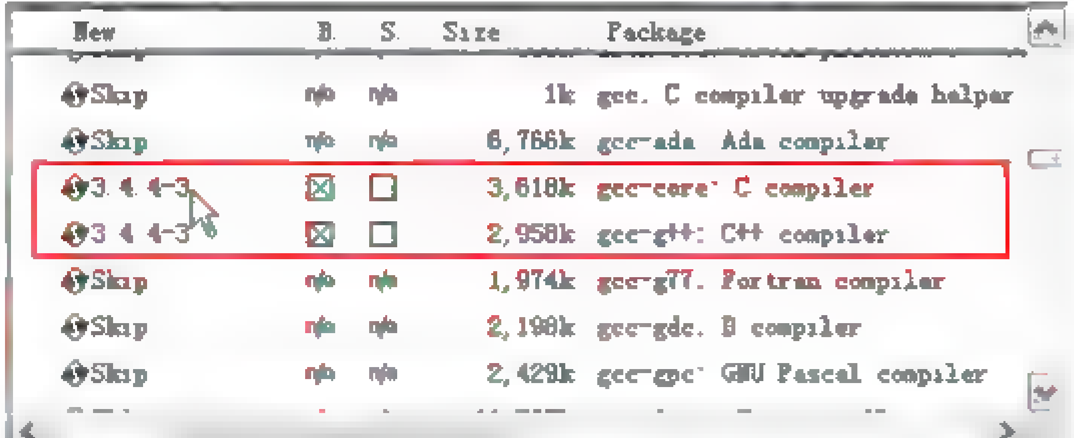


图 2-10　gcc 组件

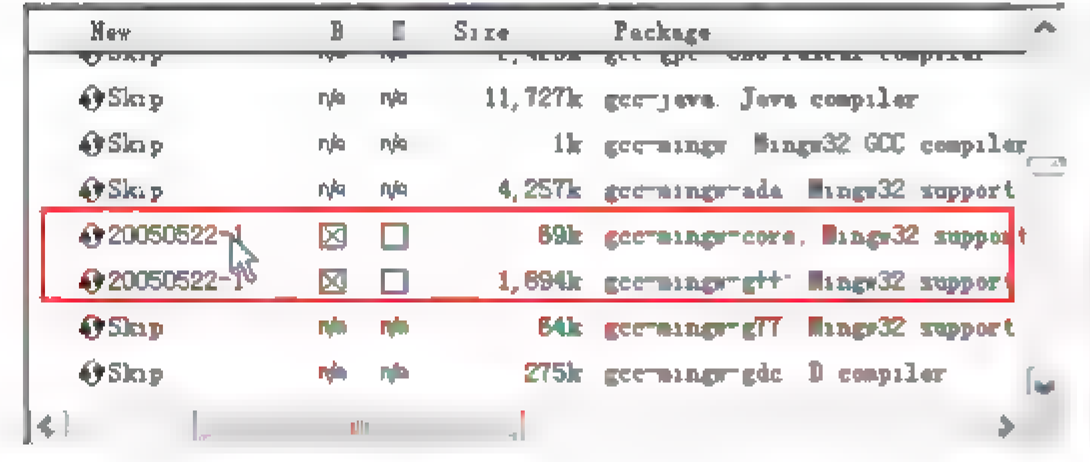


图 2-11　gcc-mingw 组件

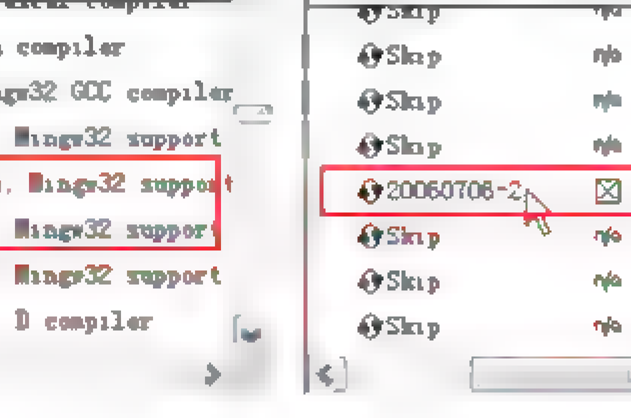


图 2-12　gdb 组件

(8) 单击“下一步”按钮进入安装过程，如图 2-13 所示。

安装的时间依据你选择的组件以及网络情况而定。安装完成后，安装程序会提示是否在桌面上创建 Cygwin 图标等，单击“完成”按钮退出安装程序，如图 2-14 所示。

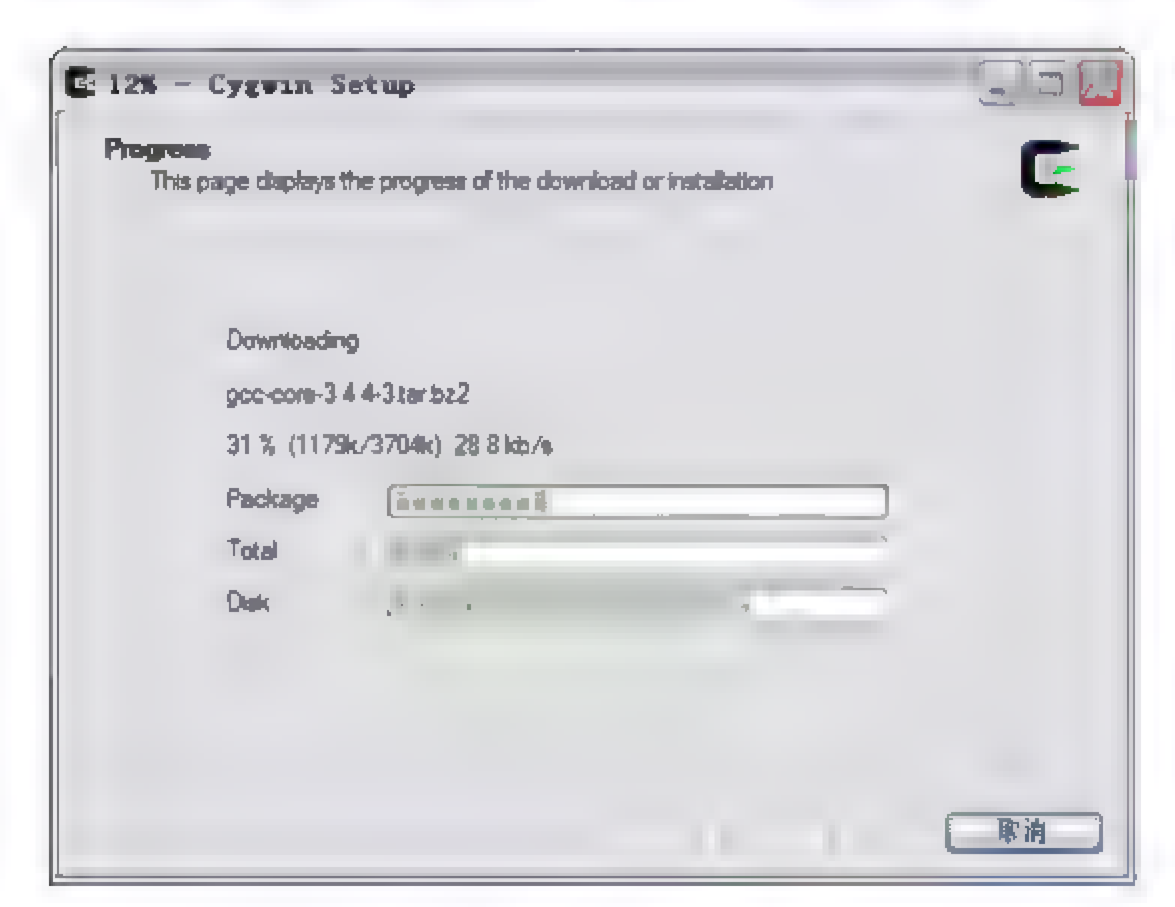


图 2-13　安装过程界面

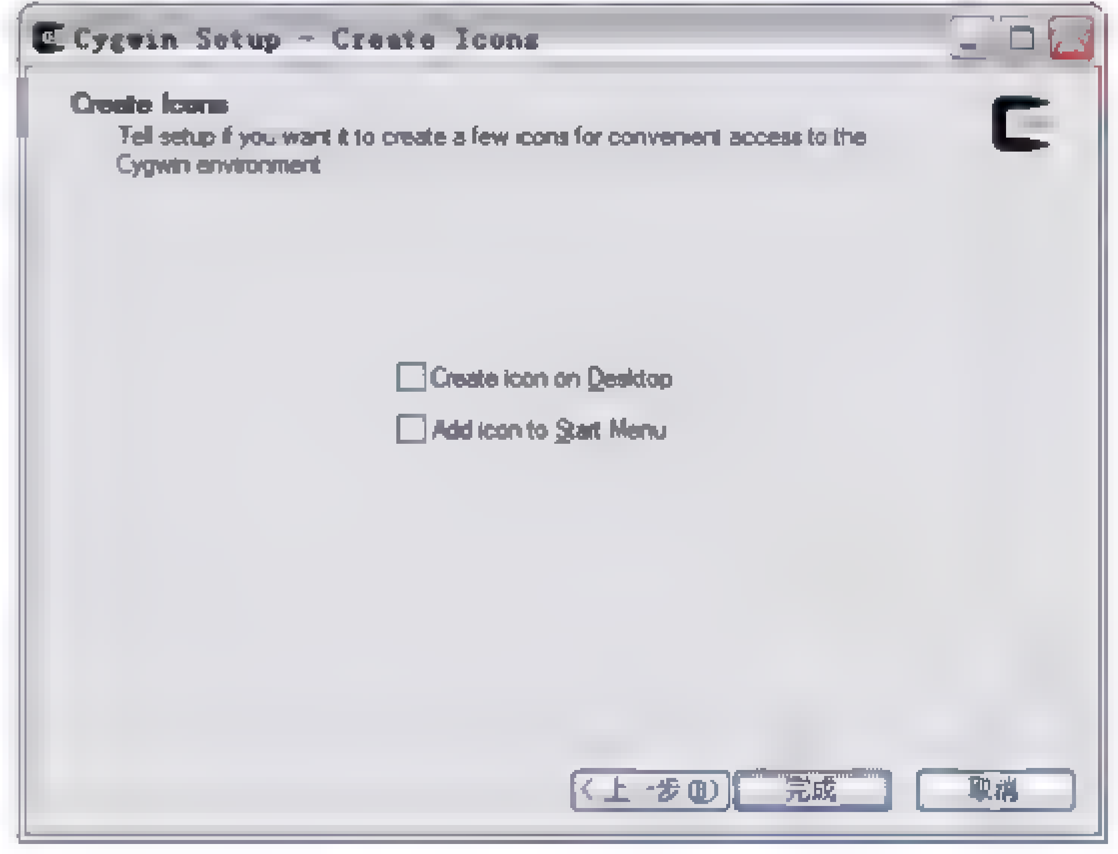


图 2-14　安装完成界面

(9) 安装完成后，接下来需要配置 Cygwin 编译器。首先启动 C-Free 进入“构建”菜单，选择“构建选项”，弹出“构建选项”对话框，如图 2-15 所示。

(10) 单击右上角的>按钮，在出现的菜单中选择“新建配置”命令，出现“新建构建配置”对话框，如图 2-16 所示。

在“编译器类型”下拉列表框中选择 CYGWIN，在“配置名称”文本框中输入名称，这里输入的是 cygwin，当然也可以输入你自己喜欢的名称。

(11) 单击“确定”按钮后弹出“编译器位置”对话框，如图 2-17 所示。

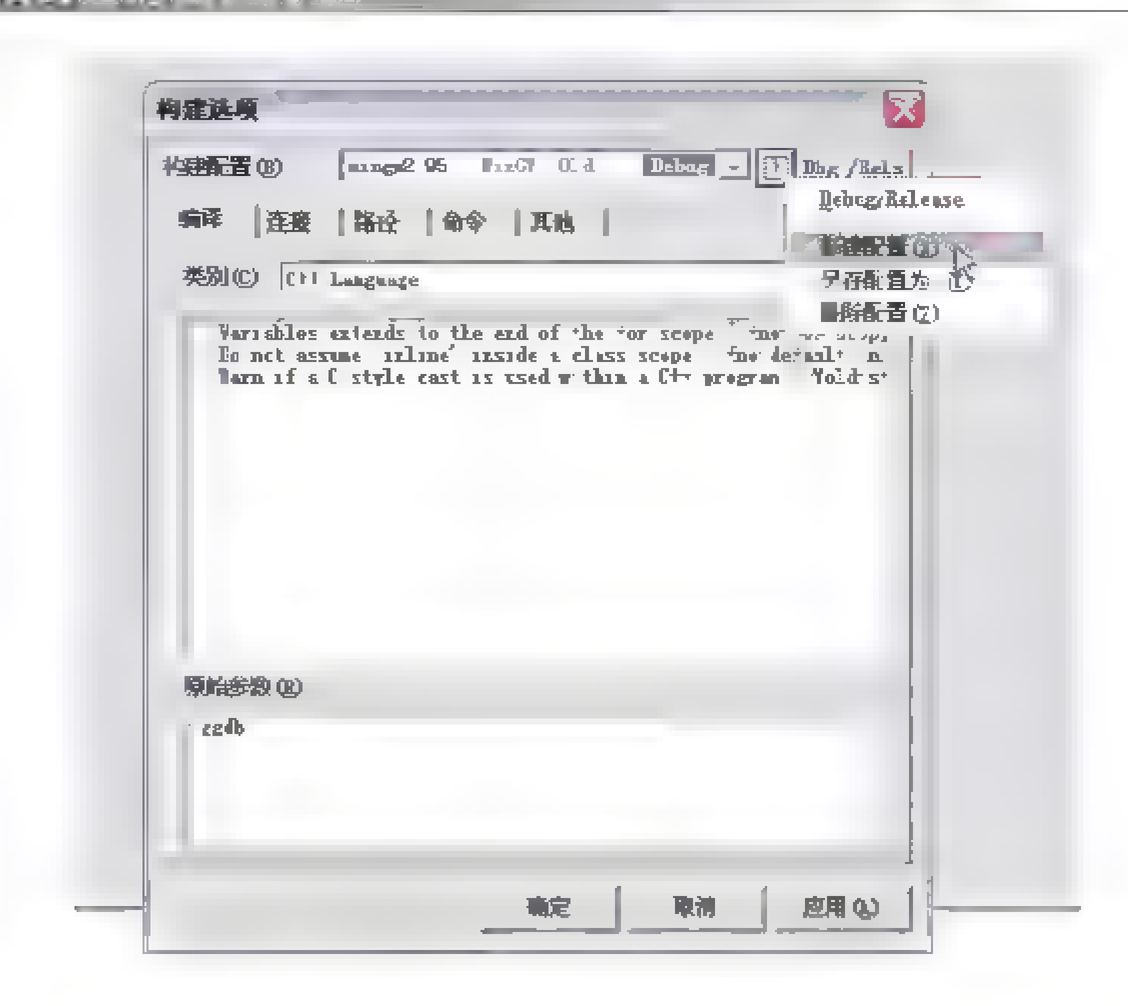


图 2-15　“构建选项”对话框

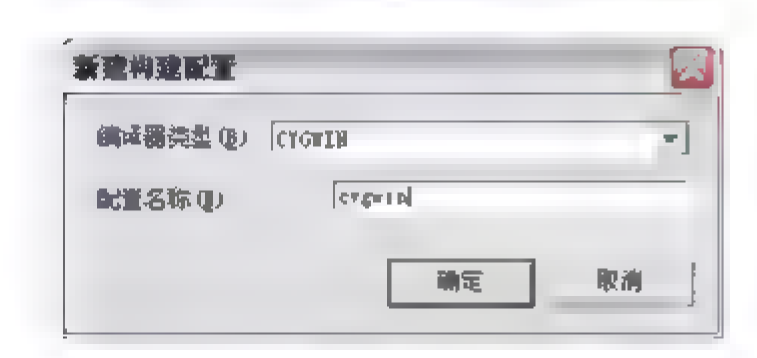


图 2-16　“新建构建配置”对话框

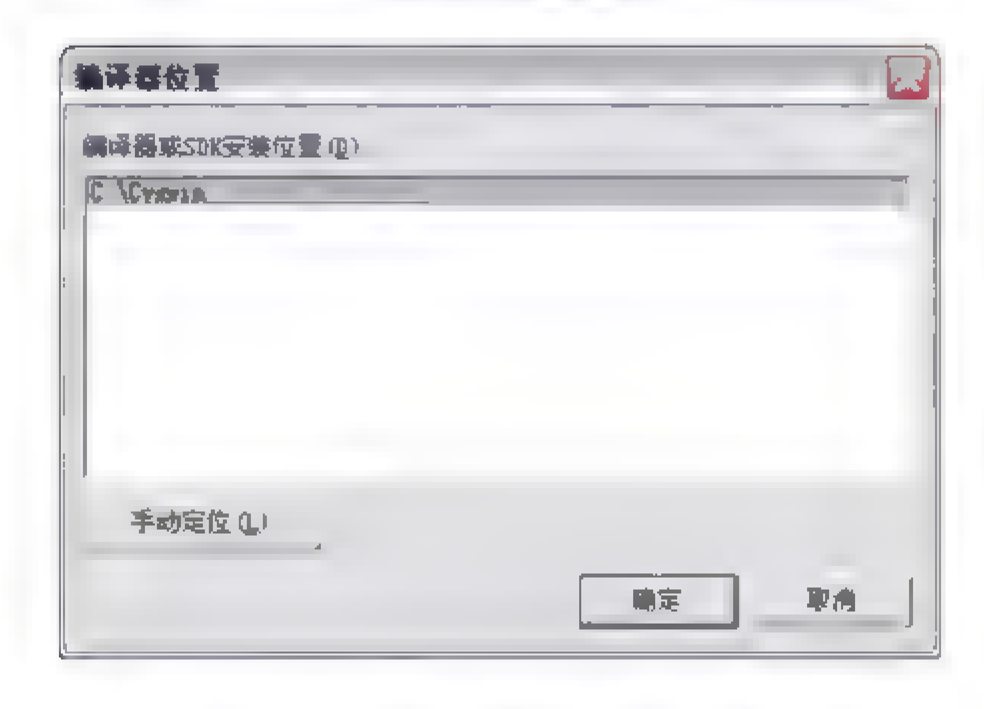


图 2-17　“编译器位置”对话框

因为在前面的安装步骤中，将 Cygwin 安装到了“C:\Cygwin\”目录下，所以在安装位置列表中会看到 C-Free 检测到了这个目录。如果 C-Free 检测不到 Cygwin 的安装目录，则需要手动定位安装目录。

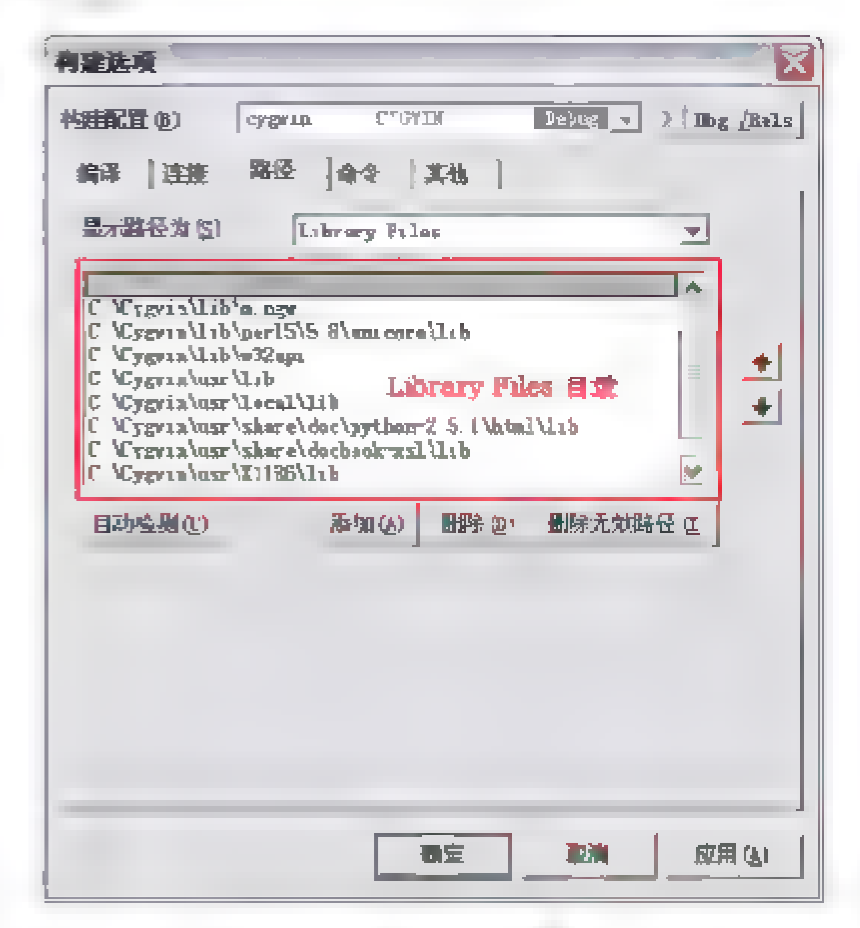
(12) 单击“确定”按钮后，编译器的各个目录就自动添加到了这个构建配置中。如图 2-18 所示，分别是添加完成后的 Include Files 目录、Library Files 目录以及 Executable Files 目录的结果截图。

如果发现 C-Free 无法自动检测出编译器的位置，原因可能是无法找到安装的编译器目录，此时需要手动添加上面的三类目录。完成后单击“确定”按钮，这样就完成了 Cygwin 编译器在 C-Free 中的配置。

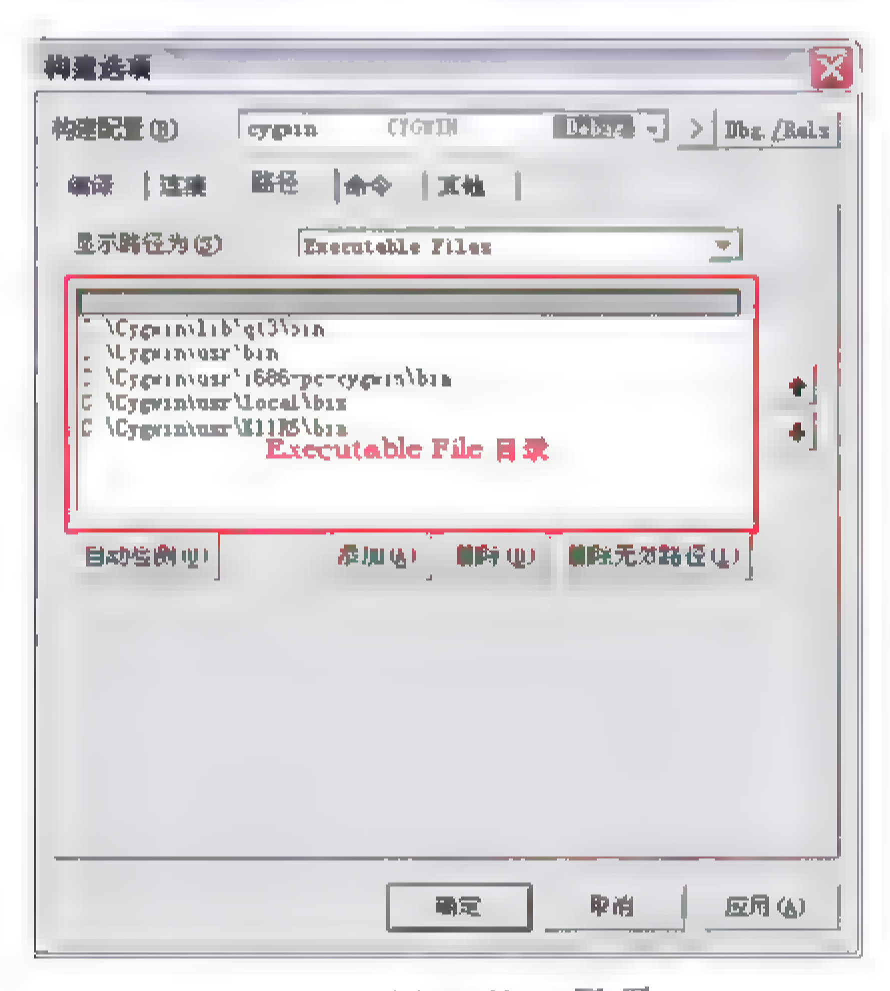
通过上面的操作，实际上是建立了一个名为“cygwin”的全局构建配置。这个构建配置既能用来构建单个文件，还能被复制为工程构建配置，用来构建整个工程。



Include Files 目录



Library Files 目录



Executable Files 目录

图 2-18　分别添加目录到构建选项值

(13) 完成了编译器在 C-Free 中的添加工作之后，就可以在 C-Free 中使用这个编译器了。只要在编译时选中刚刚添加的这个配置，后台就会使用 Cygwin 来编译，如图 2-19 所示。



图 2-19　选择 Cygwin

图 2-20 显示的是第一次启动 Cygwin 时的情况，创建主目录并执行 shell 启动文件后会有提示。现在可以运行 UNIX 命令了。

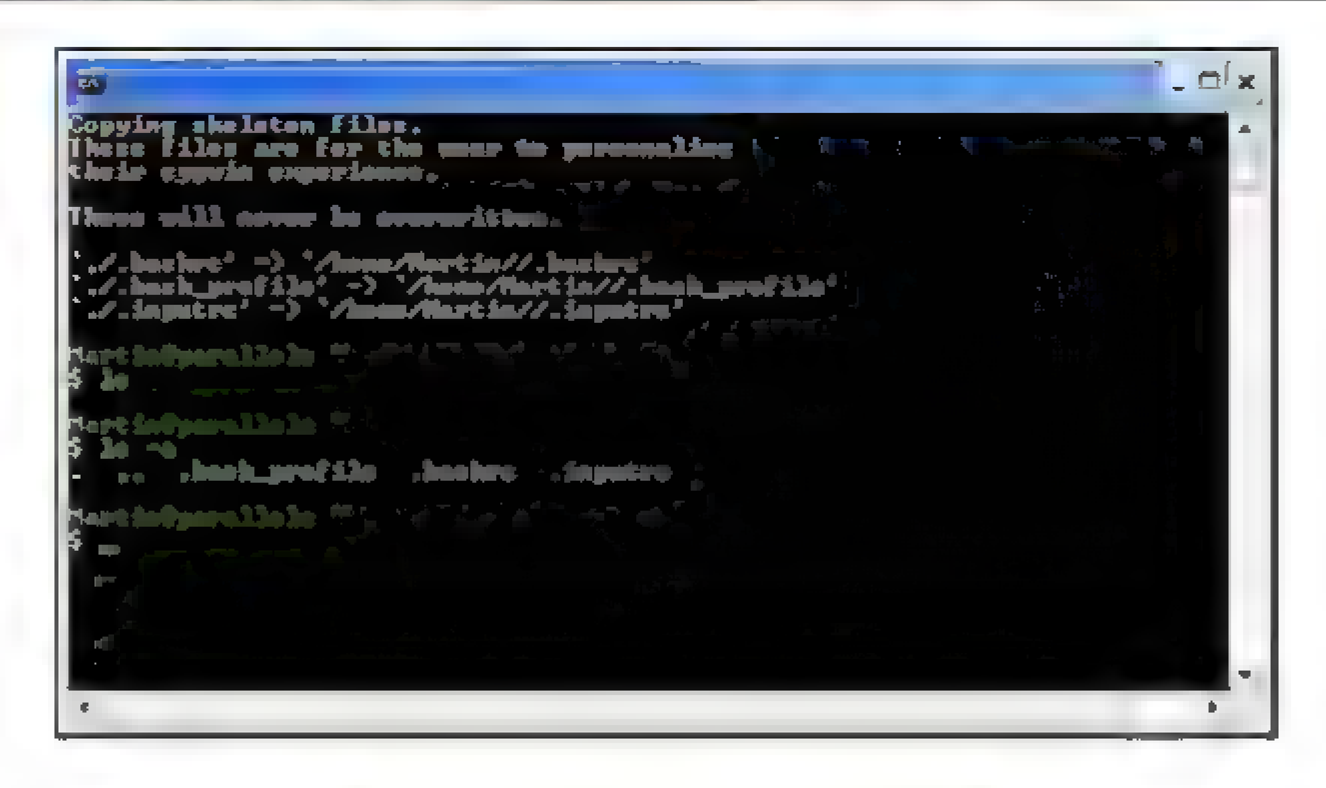


图 2-20　启动 Cygwin 时的界面

2．安装编译器

JDK 中最核心的代码(Java 虚拟机及 JDK 中 Native 方法的实现等)是使用 C++语言以及少量的 C 语言编写的，在其官方文档中声称其内部开发环境是在 Microsoft Visual Studio C++ 2003(VS2003)中进行编译的，同时也在 Microsoft Visual Studio C++ 2010(VS2010)中测试过，所以最好只选择这两个编译器之一进行编译。如果选择 VS2010，那么在编译器之中已经包含了 Windows SDK v7.0a，否则可能还要自己去下载这个 SDK，并且更新 PlatformSDK 目录。笔者在此建议读者使用 Visual Studio C++ 2010 或 Visual Studio C++ 2010 Express 进行编译。

需要特别注意的一点：Cygwin 和 VS2010 安装之后都会在操作系统的 PATH 环境变量中写入自己的 bin 目录路径，必须检查并保证 VS2010 的 bin 目录一定要在 Cygwin 的 bin 目录之前，因为这两个软件的 bin 目录之中各自都有个连接器“link.exe”，但是只有 VS2010 中的连接器可以完成 OpenJDK 的编译。

下载并安装 VS2010 的基本步骤如下。

(1) 将安装盘放入光驱，或双击存储在硬盘内的安装文件 autorun.exe，弹出安装界面，如图 2-21 所示。

(2) 单击“安装 Microsoft Visual Studio 2010”链接，弹出组件加载对话框，如图 2-22 所示。



图 2-21　开始安装界面

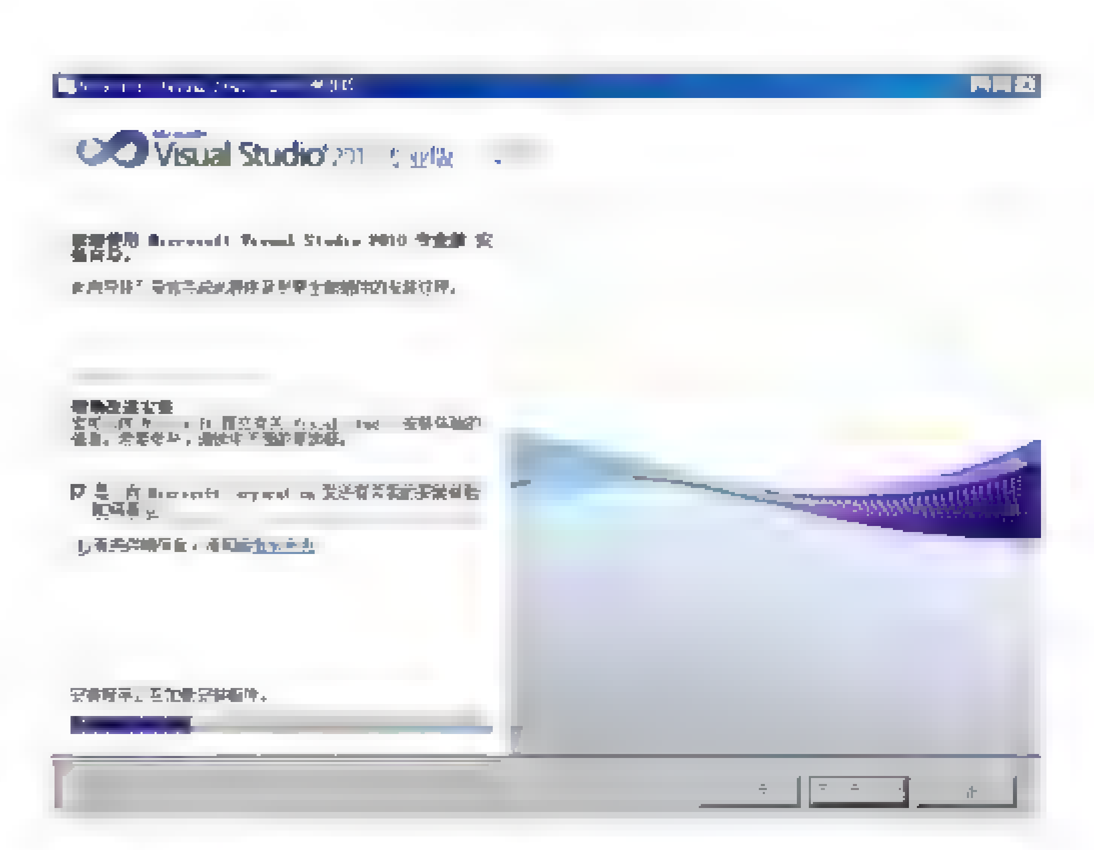


图 2-22　组件加载对话框

(3) 加载完毕后单击“下一步”按钮后弹出安装起始页界面，选中“我已阅读并接受许可条款”，如图 2-23 所示。

(4) 单击“下一步”按钮，弹出安装选项页界面，选中“完全”安装和安装路径，如图 2-24 所示。



图 2-23　安装起始页界面

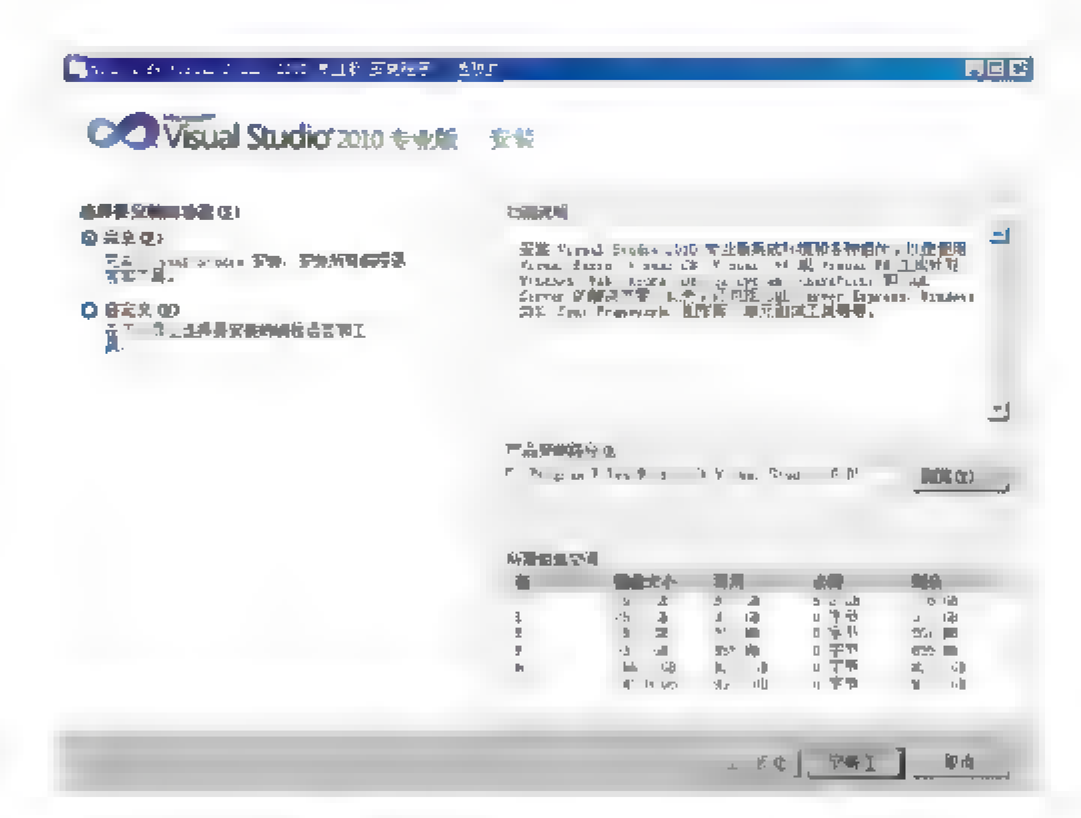


图 2-24　安装选项页界面

(5) 单击“安装”按钮后弹出安装页界面，开始进行安装，如图 2-25 所示。

(6) 安装完毕后弹出完成页界面，Visual Studio 2010 成功安装完成，如图 2-26 所示。

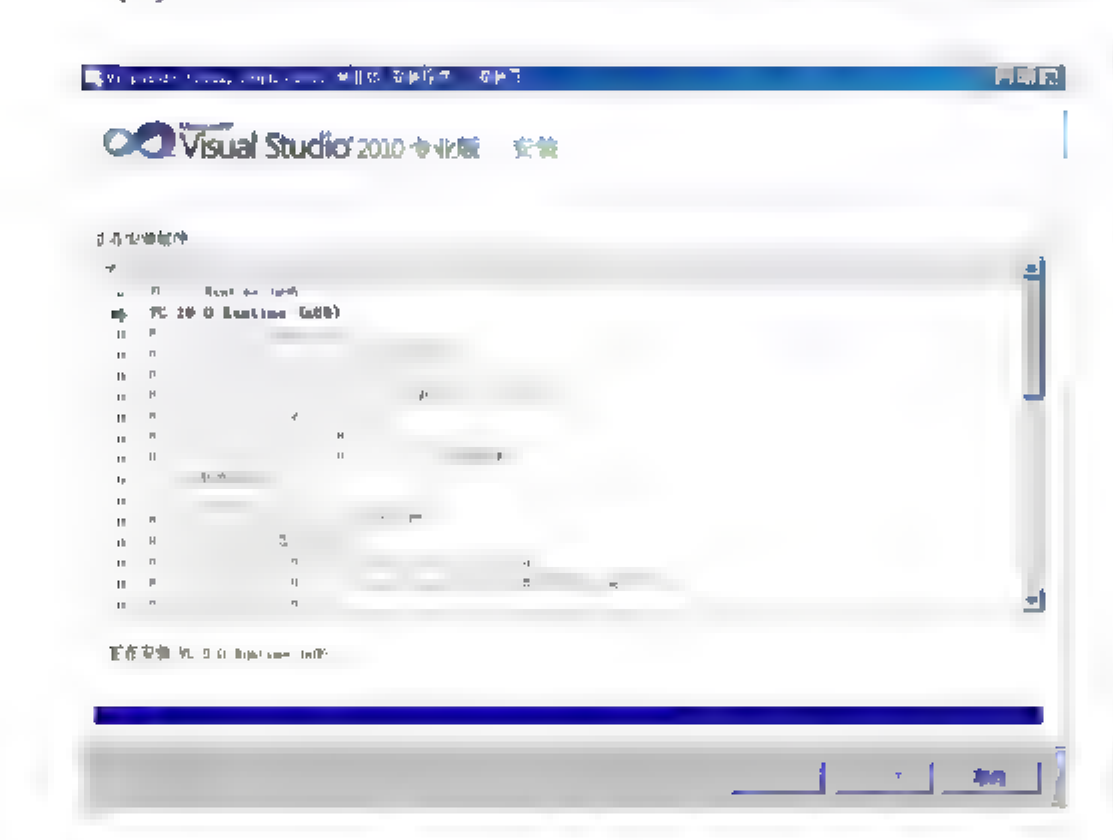


图 2-25　安装页界面

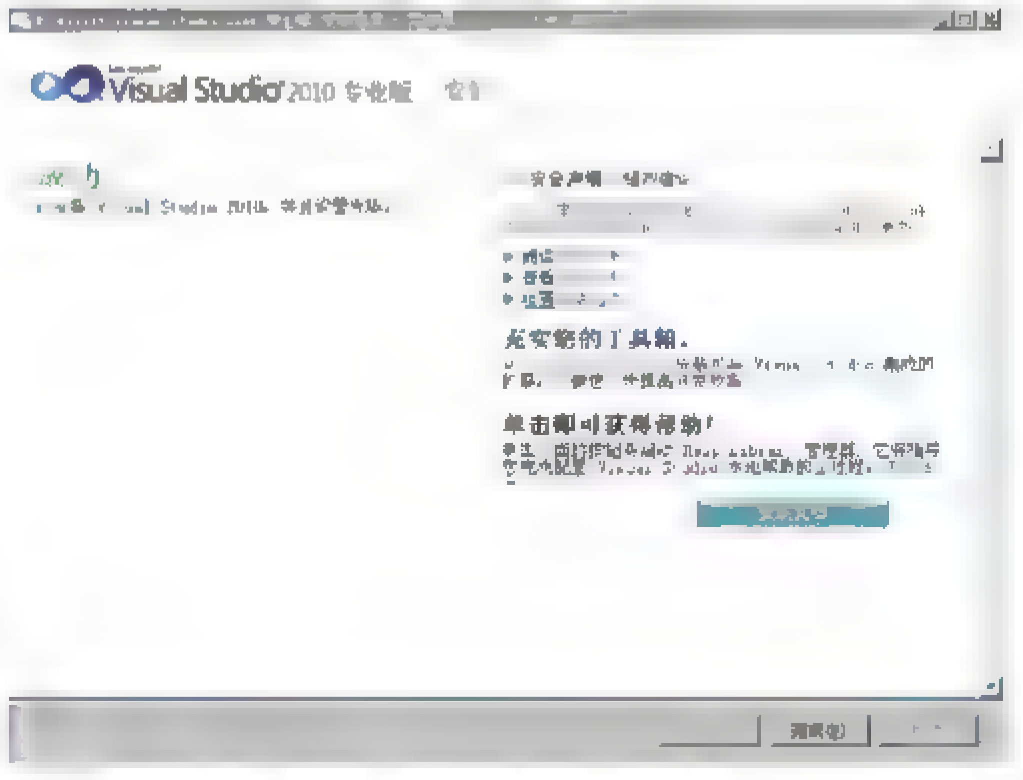


图 2-26　完成页界面

Visual Studio 2010 容量巨大，有数 G 之巨。在安装过程中一定要有耐心，慢慢等待。如果以前在机器上安装过，建议用卸载工具将原来安装的资料和痕迹完全卸载后再安装，这样会避免很多不必要的麻烦。在图 2-25 所示的安装界面中，会多次重新启动，此时不要惊慌，电脑重启后将自动进入安装界面。

另外因为需要安装很多组件，例如，数据库和 IIS 等组件，所以安装过程中总会出现这样或那样的问题。比较常见的问题是在安装 Windows 组件时，不能安装 IIS 中的 Front Page 服务器扩展，已经插入安装光盘了，却一直提示“将 XP profession service pack 2 CD 插入选定的驱动器”。这是因为 Windows 的系统文件保护不让通过，解决方法是关闭文件保护功能，关闭方法如下。

(1) 运行 gpedit.msc 打开组策略。

(2) 依次展开至计算机配置→管理模板→系统→Windows 文件保护。

(3) 找到“设置文件保护”，双击并修改为“已禁用”，然后重新启动系统就可以了。

上面的方法虽可行，但是治标不治本，还有种方法可以彻底修复 Windows 文件。插入系统安装光盘，并运行 sfc /scannow 命令检测并修复可能受损坏和更改的系统文件。这样就不会再出现此问题了。当遇到上述问题时，建议读者先试试第一个方法，再试试第二个方法。

3. 下载一个已经编译好了的 JDK

因为在 JDK 中包含的各个部分，例如，Hotspot、JDK API、JAXWS、JAXP，这些部分有的是使用 C++编写的，而更多的代码是使用 Java 语言实现的，因此编译这些 Java 代码需要用到一个可用的 JDK，官方称这个 JDK 为“Bootstrap JDK”。而编译 OpenJDK 7 的话，Bootstrap JDK 必须使用 JDK6 Update 14 或之后的版本，笔者选用的是 JDK7 Update 1。

下载并安装 JDK 的基本步骤如下。

(1) 在 Sun 官方网站下载，网址为 http://developers.sun.com/downloads/，如图 2-27 所示。

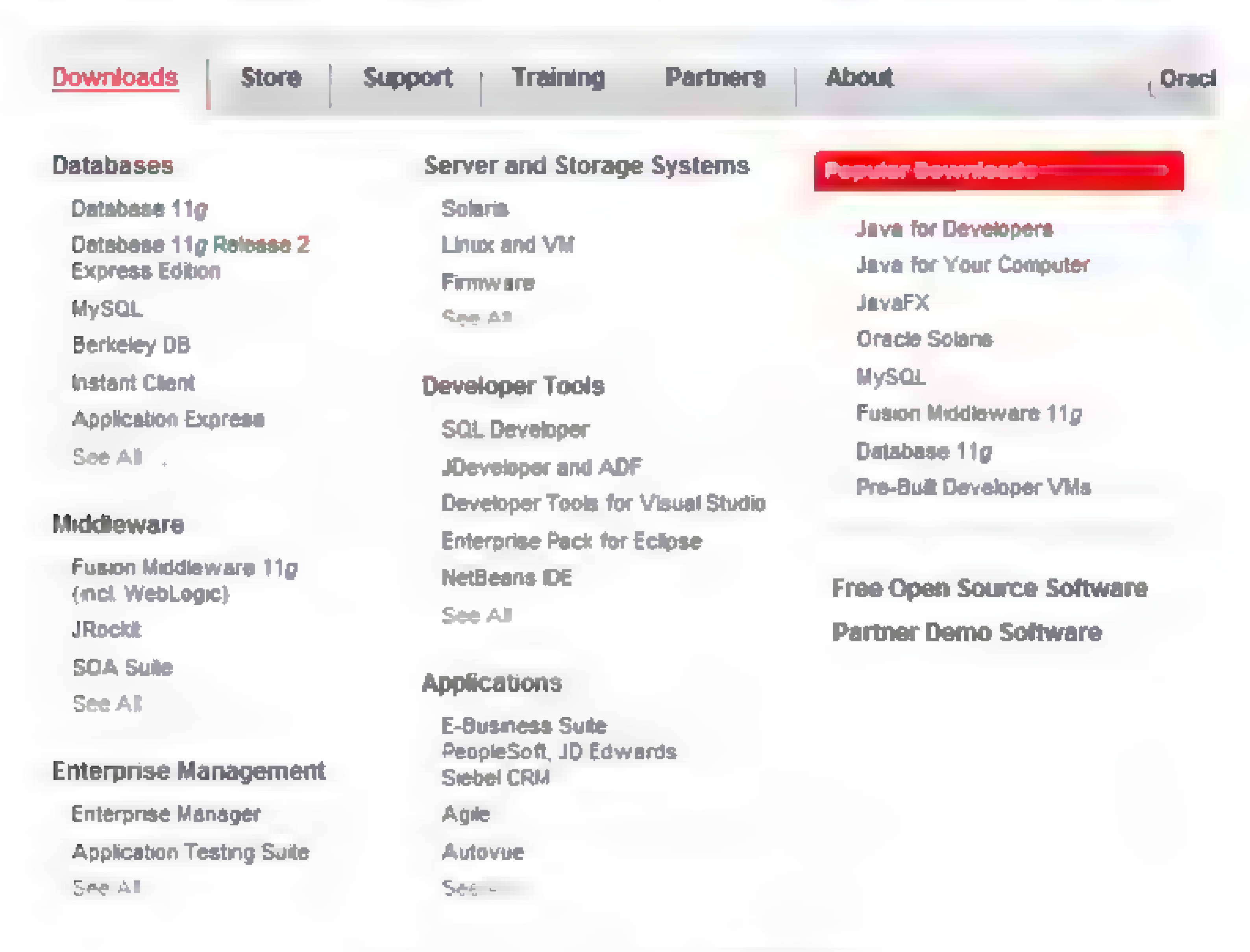


图 2-27　Sun 官方下载页面

(2) 在图 2-27 中可以看到有很多版本，在此选择当前最新的版本 Java 7，其下载页面如图 2-28 所示。

(3) 在图 2-28 中单击 JDK 下方的 Download 按钮，在弹出的新界面中选择要下载的 JDK，例如，笔者选择的是 Windows x86 版本，如图 2-29 所示。

(4) 下载完成后双击下载的“.exe”文件开始进行安装，将弹出安装向导对话框，单击“下一步”按钮，如图 2-30 所示。

(5) 弹出安装路径界面，选择文件的安装路径，如图 2-31 所示。

(6) 在此设置的安装路径是 E:\jdk1.7.0_01\，然后单击“下一步”按钮开始在安装路径解压缩下载的文件，如图 2-32 所示。

Java Platform, Standard Edition

Java SE 7u1

This release includes many security fixes. Learn more

"What Java Do I Need?" You must have a copy of the JRE (Java Runtime Environment) on your system to run Java applications and applets. To develop Java applications and applets, you need the JDK (Java Development Kit), which includes the JRE.

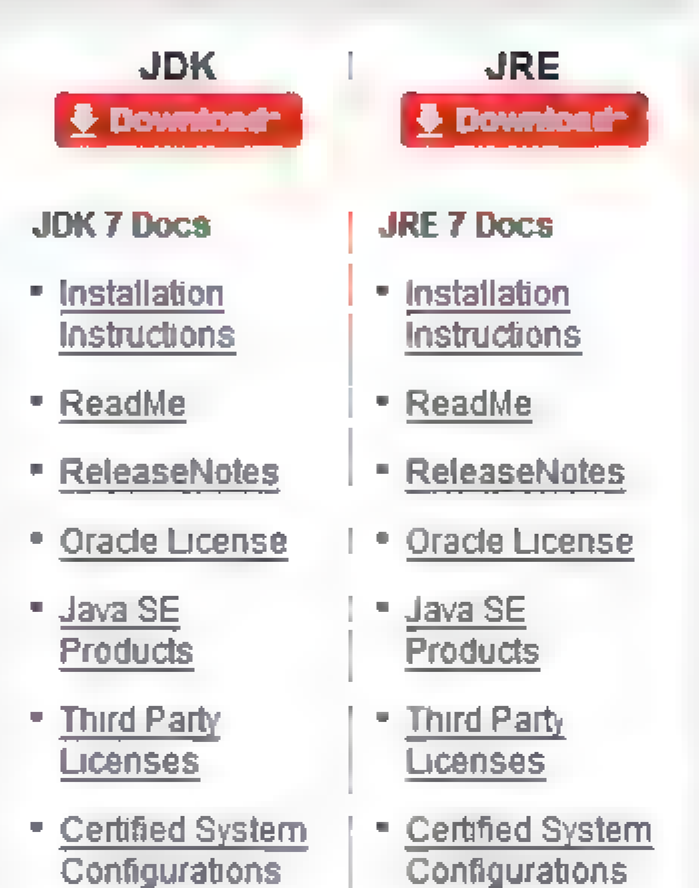


图 2-28　JDK 下载页面

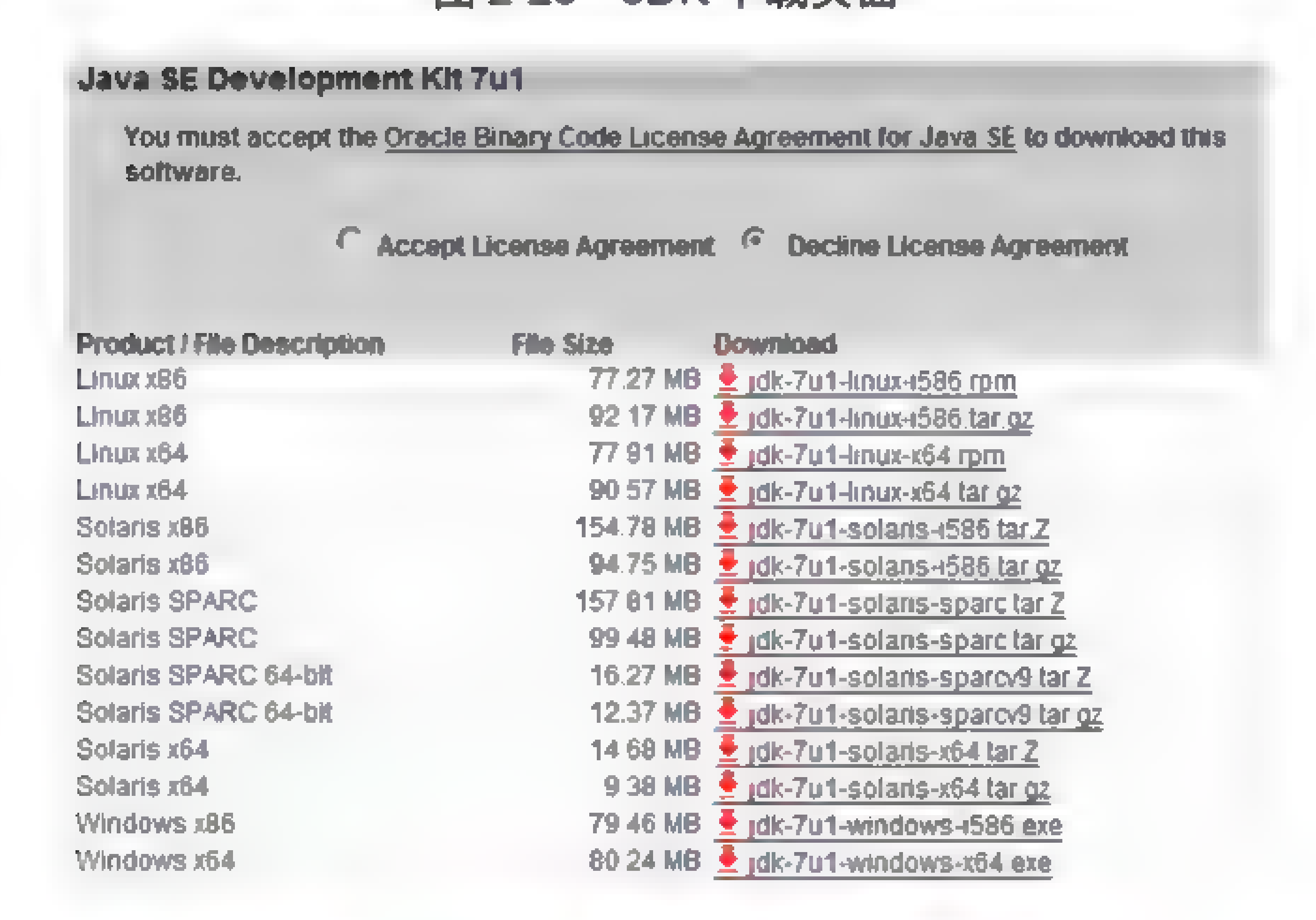


图 2-29　选择 Windows x86 版本

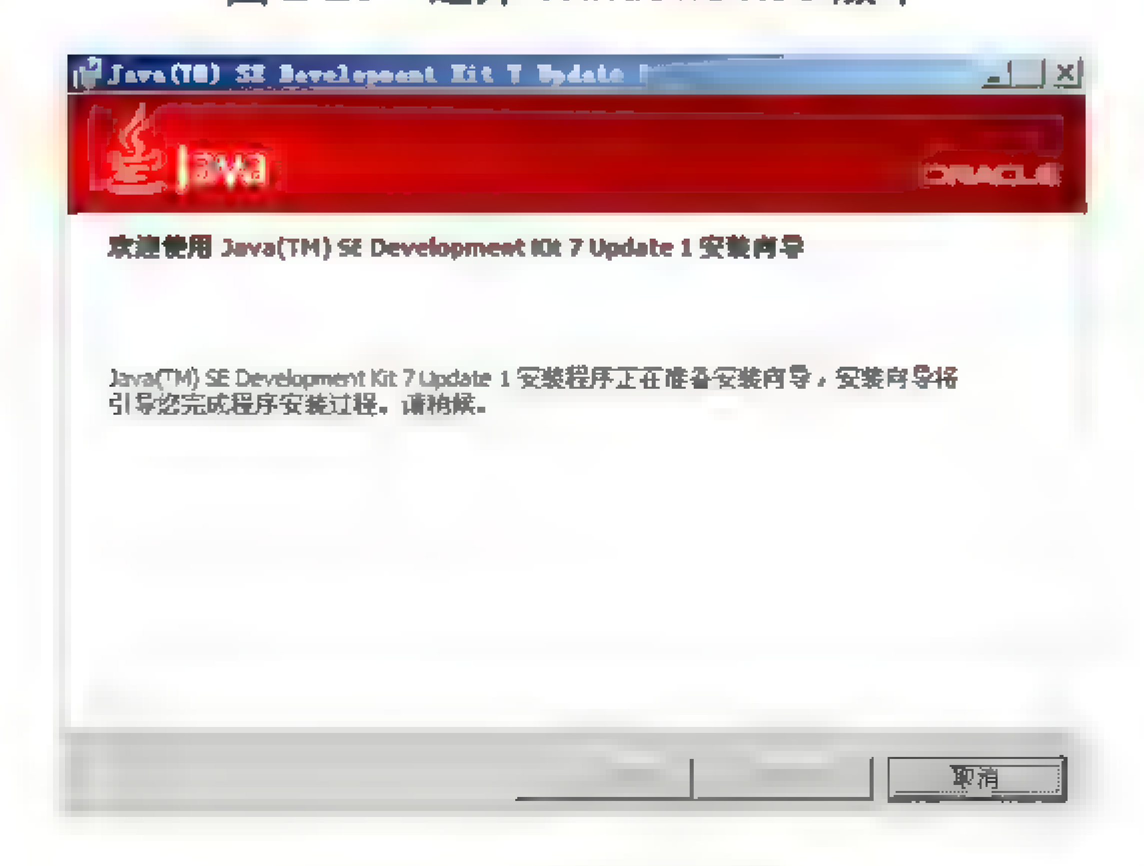


图 2-30　安装向导对话框

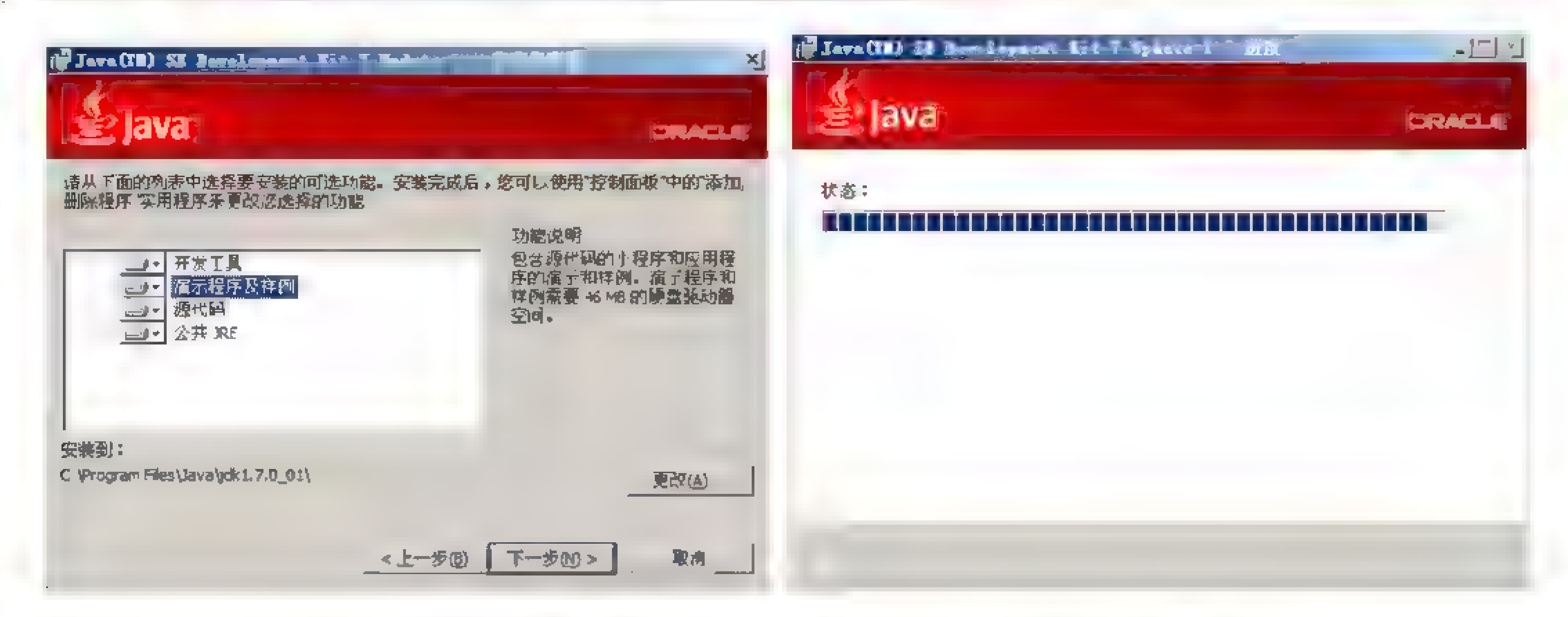


图 2-31　安装路径界面　　　　图 2-32　解压缩下载的文件

(7) 完成后弹出“目标文件夹”对话框，选择要安装的位置，如图 2-33 所示。

(8) 单击“下一步”按钮后开始正式安装，如图 2-34 所示。





图 2-33　“目标文件夹”对话框　　　　图 2-34　继续安装

(9) 安装完成后弹出“完成”对话框，单击“完成”按钮完成整个安装过程，如图 2-35 所示。



图 2-35　完成安装

安装完成后可以检测是否安装成功，检测方法是依次单击“开始”→“运行”命令，在“运行”对话框中输入“cmd”并按 Enter 键，在打开的 cmd 窗口中输入“java -version”，如果显示如图 2-36 所示的提示信息，则说明安装成功。



图 2-36　cmd 窗口

4．安装 Apache Ant

Apache Ant 是一个款将软件编译、测试、部署等步骤联系在一起加以自动化的一个工具，大多用于 Java 环境中的软件开发。由 Apache 软件基金会所提供。

Apache Ant 的用户群大多数的 Java 设计都被用于管理大量信息流，例如，纽约州就使用 Apache Ant 去管理美国最大的青年计划，每天可以实时更新超过 25 万学生的记录。

JDK 中 Java 代码部分都是使用 ANT 脚本进行编译的，在编译时要求 ANT 的版本在 1.6.5 以上。下载并安装 Apache Ant 的基本步骤如下所示。

(1) 登录 Apache Ant 等官方地址 http://ant.apache.org/bindownload.cgi 进行下载。

(2) 将下载到的文件解压到你想放置的盘符，例如 D:\Program Files\ant\apache-ant-1.8.1。

(3) 将 FB 安装路径下的 flexTasks.jar 复制到 ANT 根路径下的 lib 中。文件 flexTasks.jar 在 FB 中的路径是 C:\Program Files\Adobe\Adobe Flash Builder 4.0.0\ant\lib。ANT 下的 lib 路径是 D:\Program Files\ant\apache-ant-1.8.1\lib。

(4) 配置环境变量。

变量：ANT_HOME。

值：是刚解压到的路径，即 D:\Program Files\ant\apache-ant-1.8.1。

设置 Path：%ANT_HOME%\bin。

到此为止，ANT 就安装完成了，运行 cmd，输入 ANT，如果没有指定 build.xml 则会输出：

```
Buildfile: build.xml does not exist!
Build failed
```

例如：

```
C:\Documents and Settings\Administrator>ant
Buildfile: build.xml does not exist!
Build failed
```

2.2.5　准备依赖项

在具体编译之前，还需要准备下面的依赖项。

1．安装 JDK Plug

OpenJDK 中的源码并没有 100%开源，还有少量部分的无法开源的产权代码。虽然 OpenJDK 承诺日后将逐步开源这部分产权代码，但现在编译 JDK 还需要这部分闭源包，官方称之为“JDK Plug”，它们从前面的 Source Releases 页面就可以下载到。在 2011 以前的版本中，还需要使用 JDK plug，在之后的版本中就不需要 JDK plug 了。

在 Windows 平台的 JDK Plug 是以 Jar 包的形式提供的，通过如下命令可以将其安装：

```
java -jar jdk-7-ea-plug-b121-windows-i586-09 dec 2010.jar
```

运行后将会显示如图 2-37 所示的协议，单击 ACCEPT 按钮接受协议，然后把 Plug 安装到指定目录。安装完毕后建立一个名为 ALT_BINARY_PLUGS_PATH 的环境变量，变量值为此 JDK Plug 的安装路径，后面编译程序时需要用到它。

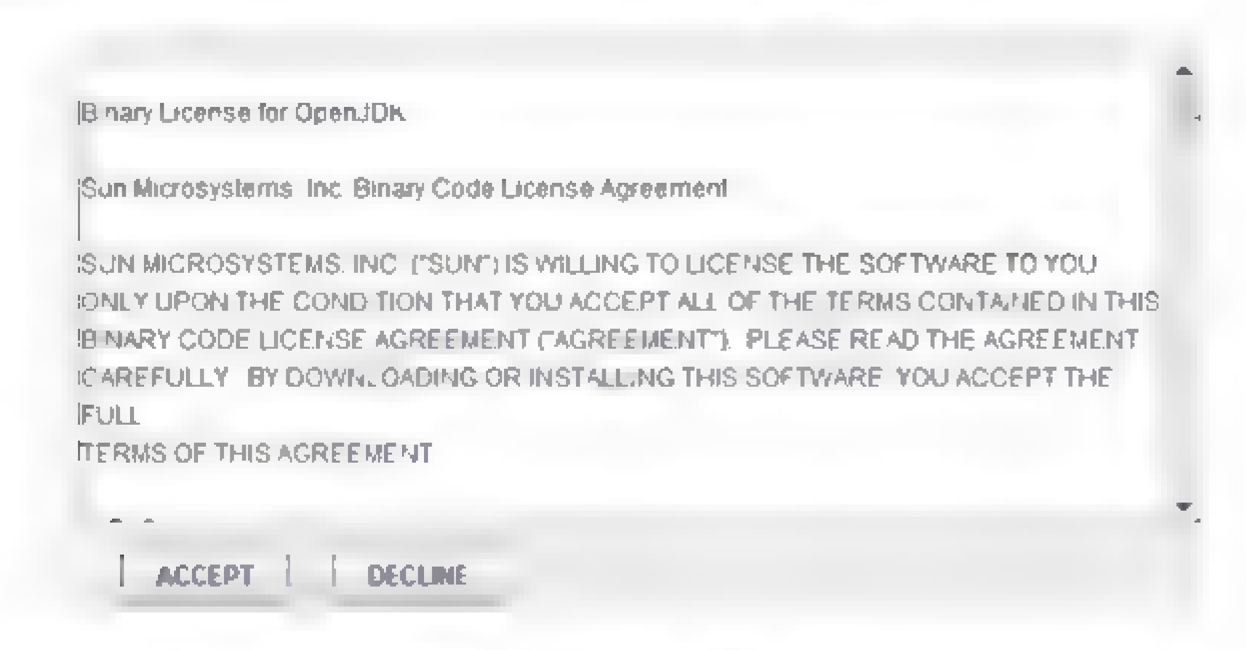


图 2-37　协议界面

2．引用 JDK 的运行时包

除了要用到 JDK Plug 外，编译时还需要引用 JDK 的运行时包，用 Java 语言编写的部分需要用到这个包。如果仅仅是想编译一个 HotSpot 虚拟机，则可以不用这个包。官方文档把这部分称之为“Optional Import JDK”，可以直接使用前面 Bootstrap JDK 的运行时包，我们需要建立一个名为 ALT_JDK_IMPORT_PATH 的环境变量，并且这个变量指向 JDK 的安装目录。

3．安装 FreeType 库

FreeType 库是一个完全免费(开源)的、高质量的且可移植的字体引擎，它提供统一的接口来访问多种字体格式文件，包括 TrueType、OpenType、Type1、CID、CFF、Windows FON/FNT、X11 PCF 等。FreeType 支持单色位图、反走样位图的渲染，是一个高度模块化的程序库，虽然它使用 ANSI C 开发，但是采用面向对象的思想，所以 FreeType 用户可以灵活地对它进行裁剪。

为了编译 JDK，我们需要安装一个 2.3 以上版本的 FreeType，JDK 的 Swing 部分和 JConsole 这类工具要使用到它。安装好后建立两个环境变量 ALT_FREETYPE_LIB_PATH 和 ALT_FREETYPE_HEADERS_PATH，分别指向 FreeType 安装目录下的 bin 目录和 include 目录，另外，还需要把 FreeType 的 bin 目录加入到 PATH 环境变量中。

4．安装 Microsoft DirectX 9.0 SDK

微软的 DirectX 包括加速视频卡和声卡驱动程序，能够为不同类型的多媒体提供更好的播放效果，例如，全色图形、图像、三维动画、音乐，以及剧场声音。DirectX 使用这些高级功能而不要求识别计算机中的硬件组件，并确保大多数软件可以在大部分硬件系统上运行。DirectX 由应用程序编程接口(API)组成，又分成 DirectX 基础层和 DirectX 媒体层两类。这些 API 可以让程序直接访问计算机的许多硬件设备。

读者可以登录微软的官方网站下载 Microsoft DirectX 9.0 SDK，它是完全免费的。安装后建立环境变量 ALT_DXSDK_PATH 指向 DirectX 9.0 SDK 的安装目录。

5．使用 MSVCR100.DLL 动态链接库

在编译时需要一个名为“MSVCR100.DLL”的动态链接库，如果已经安装了 Visual Studio 2010，那么在本机就能找到这个文件。找到后新建环境变量 ALT_MSVCRNN_DLL_PATH 指向这个文件所在的目录。如果读者选择的是 VS2003，则此文件名为 MSVCR73.DLL，可以从网络资源中获取。

2.2.6　开始编译

经过本章前面步骤的准备之后，接下来就可以进行具体的编译工作了。具体的编译步骤如下。

(1) 执行 VS2010 中的 VCVARS32.BAT，这个批处理文件的目的主要是设置 INCLUDE、LIB 和 PATH 这几个环境变量，如果和笔者一样只是下载了编译器的话则需要手工设置它们，各个环境变量的设置值可以参考下面步骤(3)后面的代码内容。批处理运行完之后建立 ALT_COMPILER_PATH 环境变量让 Makefile 知道在哪里可以找到编译器。

(2) 分别建立名为 ALT_BOOTDIR 和 ALT_JDK_IMPORT_PATH 的两个环境变量，它们指向前面提到的 JDK 7 的安装目录。

(3) 建立 ANT_HOME 指向 Apache ANT 的安装目录。整个过程需要建立很多环境变量，各个变量的具体说明如下所示。

```
SET ALT BOOTDIR=D:/ DevSpace/JDK 1.7.0 1
SET ALT BINARY PLUGS PATH=D:/jdkBuild/jdk7plug/openjdk-binary-plugs
SET ALT JDK IMPORT PATH=D:/ DevSpace/JDK 1.7.0 01
SET ANT HOME=D:/jdkBuild/apache-ant-1.8.1
SET ALT MSVCRNN DLL PATH=D:/jdkBuild/msvcr100
SET ALT DXSDK PATH=D:/jdkBuild/msdxsdk
SET ALT COMPILER PATH=D:/jdkBuild/vcpp2010.x86/bin
SET ALT FREETYPE HEADERS PATH=D:/jdkBuild/freetype-2.3.5-1-bin/include
SET ALT FREETYPE LIB PATH=D:/jdkBuild/freetype-2.3.5-1-bin/bin

SET INCLUDE=D:/jdkBuild/vcpp2010.x86/include;D:/jdkBuild/vcpp2010.x86/
sdk/Include;%INCLUDE%
SET LIB=D:/jdkBuild/vcpp2010.x86/lib;D:/jdkBuild/vcpp2010.x86/sdk/Lib;%LIB%
SET LIBPATH=D:/jdkBuild/vcpp2010.x86/lib;%LIB%
SET PATH=D:/jdkBuild/vcpp2010.x86/bin;D:/jdkBuild/vcpp2010.x86/dll/
x86;D:/Software/OpenSource/Cygwin/bin;%ALT_FREETYPE_LIB_PATH%;%PATH%
```

上述各个软件都是前面下载的，具体版本号是笔者写稿时的版本，读者可以根据实际情况替换为自己软件的版本号。

(4) 取消环境变量 JAVA HOME。

(5) 接下来正式开始进行编译工作。进入控制台(Cmd.exe)后运行刚才准备好的设置环境变量的批处理文件，然后输入“bash”进入 Bourne Again Shell 环境。输入“make sanity”来检查我们前面所做的设置是否全部正确。如果 JDK 的安装源码中存在“jdk_generic profile.sh”Shell 脚本，则需要先执行它。如果一切顺利则会出现类似下面的输出。

```
D:\jdkBuild\openjdk7>bash
bash-3.2$ make sanity
Cygwin warning:
  MS-DOS style path detected: C:/Windows/system32/wscript.exe
  Preferred POSIX equivalent is: /cygdrive/c/Windows/system32/wscript.exe
  Cygwin environment variable option "nodosfilewarning" turns off this
warning.
  Consult the user's guide for more details about POSIX paths:
    http://Cygwin.com/Cygwin-ug-net/using.html#using-pathnames
( cd  ./jdk/make && \
…
OpenJDK-specific settings:
  FREETYPE HEADERS PATH = D:/jdkBuild/freetype-2.3.5-1-bin/include
    ALT FREETYPE HEADERS PATH = D:/jdkBuild/freetype-2.3.5-1-bin/include
  FREETYPE LIB PATH = D:/jdkBuild/freetype-2.3.5-1-bin/bin
    ALT FREETYPE LIB PATH = D:/jdkBuild/freetype-2.3.5-1-bin/bin

OPENJDK Import Binary Plug Settings:
  IMPORT BINARY PLUGS = true
  BINARY PLUGS JARFILE = D:/jdkBuild/jdk7plug/openjdk-binary-
plugs/jre/lib/rt-closed.jar
    ALT BINARY PLUGS JARFILE =
  BINARY PLUGS PATH = D:/jdkBuild/jdk7plug/openjdk-binary-plugs
    ALT BINARY PLUGS PATH = D:/jdkBuild/jdk7plug/openjdk-binary-plugs
  BUILD BINARY PLUGS PATH =
J:/re/jdk/1.7.0/promoted/latest/openjdk/binaryplugs
    ALT BUILD BINARY PLUGS PATH =
  PLUG LIBRARY NAMES =

Previous JDK Settings:
  PREVIOUS RELEASE PATH = USING-PREVIOUS RELEASE IMAGE
    ALT PREVIOUS RELEASE PATH =
  PREVIOUS JDK VERSION = 1.6.0
    ALT PREVIOUS JDK VERSION =
  PREVIOUS JDK FILE =
    ALT PREVIOUS JDK FILE =
  PREVIOUS JRE FILE =
    ALT PREVIOUS JRE FILE =
  PREVIOUS RELEASE IMAGE = D:/ DevSpace/JDK 1.7.0 1
    ALT PREVIOUS RELEASE IMAGE =
Sanity check passed.
```

Makefile 的 Sanity 检查过程会输出编译所需的所有环境变量，如果输出“Sanity check passed.”，则说明检查过程通过了。此时就可以输入“make”执行整个 Makefile，如果失

败则需要根据系统输出的失败原因，回头再检查一下对应的设置。并且最好在下一次编译之前先执行“make clean”来清理掉上次编译遗留的文件。

(6) 编译成功之后，打开 OpenJDK 源码下的 build 目录，会发现新增了一个编译好的 JDK 文件。双击打开此文件，然后执行“java -version”，可以看到以自己机器命名的 JDK。

注意：为了提高编译的成功率，建议大家尽量在英文版本的操作系统上进行编译工作。如果不能在英文的系统上编译，可以尝试把系统的文字格式调整为“英语(美国)”，在“控制面板”→“区域和语言选项”的第一个选项卡中进行设置即可。

2.3　在 Linux 平台编译 JDK

本章前面已经详细讲解了在 Windows 平台编译 JDK 的步骤。本节将详细讲解基于 XUbuntu 10.10 平台，以 JRL 源码构建 JDK 6 update 23 的基本步骤。

(1) 首先获取 JDK 的最新源码，本节使用的是以下两个文件包。

- jdk-6u23-fcs-bin-b05-jrl-12_nov_2010.jar
- jdk-6u23-fcs-src-b05-jrl-12_nov_2010.jar

(2) 依次安装如下软件包。

- build-essential
- gawk
- m4
- openjdk-6-jdk
- libasound2-dev
- libcups2-dev
- libxrender-dev
- xorg-dev
- xutils-dev
- x11proto-print-dev

下面是对应的安装命令：

```
$ sudo apt-get install build-essential gawk m4 openjdk-6-jdk libasound2-dev libcups2-dev libxrender-dev xorg-dev xutils-dev x11proto-print-dev
```

上述命令的具体说明如下：

- build-essential、gawk、m4、binutils：是 Linux 上的一些基本工具，build 许多东西都需要它们。build-essential 主要用来装 g++(GNU C++编译器)及 C++标准库；gawk 是 GNU 版 awk，用来做文本操作；m4 是一种模板语言，AWT 的 DebugHelper 依赖它来生成部分源码；binutils 主要是链接器、汇编器、反汇编器之类的。
- openjdk-6-jdk：要想 build JDK，需要先安装一个启动用的 JDK(Bootstrap JDK)。

apt-get 默认将它安装在/usr/lib/jvm/java-6-openjdk 目录中。build 的时候要把 ALT_BOOTDIR 设置到它的安装目录上。这里装 Sun JDK 来做 Bootstrap JDK 也行。

- libasound2-dev：是 Advanced Linux Sound Architecture (ALSA)相关的依赖。
- libcups2-dev：是 Common UNIX Printing System (CUPS)相关的依赖。
- libxrender-dev、xorg-dev、xutils-dev、x11proto-print-dev：都是相关的依赖，主要功能是编译出 AWT 的部分。在安装完这部分后，还需要专门设置一个符号链接，把"/usr/lib"映射为"/usr/X11R6"的别名，这样编译过程才能正确找到 X11 的头文件：

```
$ sudo mkdir /usr/X11R6
$ cd /usr/X11R6/
$ sudo ln -s -T /usr/lib lib
```

注意：如果编译过程中有意外错误发生，则需要安装 binutils，下面是对应的命令：

```
$ sudo apt-get install binutils
```

(3) 将如下源码文件解压到同一目录下，确保该目录有读写权限，并确认该目录有 4G 以上的剩余空间可用。

- jdk-6u23-fcs-src-b05-jrl-12_nov_2010.jar
- jdk-6u23-fcs-bin-b05-jrl-12_nov_2010.jar

例如，笔者将源码包是解压到了/media/JDK6_build_area/jdk6u23 目录中。

(4)解决缺少符号问题，在解压出来的源码目录中的文件 j2se/src/solaris/native/sun/awt/awt_GraphicsEnv.c 中找到如下代码：

```
if (xerr->minor_code == X_ShmAttach) {
```

将 X_ShmAttach 改为 1，即修改后变为：

```
if (xerr->minor_code == 1) {
```

(5) 开始设置环境变量，在此编译过程中只需要设置一个环境变量。

```
$ export LANG=C
```

如果环境中已经设置了 JAVA_HOME，则需要用如下命令删除此环境变量，否则在编译过程中会有异常发生。

```
$ unset JAVA_HOME
```

(6) 开始检查编译环境的正确性。使用 make dev-sanity 命令即进入到解压后源码包的 control/make 目录中，在该命令后面需要带上一些环境参数，完整命令如下：

```
$ make dev-sanity BUILD_DEPLOY=false SKIP_COMPARE_IMAGES=true
ALT_BOOTDIR=/usr/lib/jvm/java-6-openjdk ALT_DEVTOOLS_PATH=/usr/bin
HOTSPOT_BUILD_JOBS=2
```

下面是对上述命令中主要参数的说明：

- dev(或者 DEV_ONLY)：设置为 true 可以让另外三个变量 SKIP_COMPARE

IMAGES、BUILD_INSTALL 和 NO_DOCS 也变为 true。

- BUILD_DEPLOY：设置为 false 可以避开 javaws 和浏览器 Java 插件之类的部分的编译。
- BUILD_INSTALL：设置为 false 就不会编译出安装包。因为安装包里有些奇怪的依赖，但即便不编译出它也已经能得到完整的 JDK 映像，所以建议别编译它。
- SKIP_COMPARE_IMAGES：能够比较本次编译出来的映像与先前版本的差异。这个对我们来说没有意义，建议直接设置为 false，否则 sanity 检查会报缺少先前版本 JDK 的映像。如果有设置 dev 或者 DEV_ONLY=true 的话，可以不显式设置。
- ALT_BOOTDIR：表示 Bootstrap JDK 的安装路径，必须设置。
- ALT_DEVTOOLS_PATH：表示 zip 和 unzip 工具所在的路径，必须设置。
- HOTSPOT_BUILD_JOBS：主要用于设置编译 HotSpot 的过程的并发程度，基本上设到跟 CPU 的核数一样多，也可以不设置。
- ALT_JDK_IMPORT_PATH：不用设置此变量。因为要编译的 JDK 足够完整，缺少的部分我们都不需要。

下面是检查过程的完整输出日志：

```
$ make dev-sanity BUILD_DEPLOY=false SKIP_COMPARE_IMAGES=true
ALT_BOOTDIR=/usr/lib/jvm/java-6-openjdk ALT_DEVTOOLS_PATH=/usr/bin
HOTSPOT_BUILD_JOBS=2
cd  ../../control/make
make sanity DEV_ONLY=true
make[1]: Entering directory
`/media/JDK6_build_area/jdk6u23/control/make'
make[2]: Entering directory `/media/JDK6_build_area/jdk6u23/j2se/make'
make[2]: Leaving directory `/media/JDK6_build_area/jdk6u23/j2se/make'

Build Machine Information:
   build machine =

Build Directory Structure:
   CWD = /media/JDK6_build_area/jdk6u23/control/make
   TOPDIR = ../..
   CONTROL_TOPDIR = ../../control
   HOTSPOT_TOPDIR = ../../hotspot
   J2SE_TOPDIR = ../../j2se
   MOTIF_TOPDIR = ../../motif

Build Directives:
   BUILD_HOTSPOT = true
   BUILD_MOTIF   = true
   BUILD_J2SE    = true
   BUILD_DEPLOY  = false
   BUILD_INSTALL = false
……………………(在此省略部分日志)
WARNING: Your are not building DEPLOY workspace from
         the control build. This will result in a development-only
         build of the J2SE workspace, lacking the plugin and javaws
```

```
binaries.

WARNING: Your are not building INSTALL workspace from
        the control build. This will result in a development-only
        build of the J2SE workspace, lacking the installation bundles

WARNING: Your build environment has the variable DEV ONLY
        defined. This will result in a development-only
        build of the J2SE workspace, lacking the documentation
        build and installation bundles.

WARNING: The official linux builds use OS version 2.4.9-e.3.
        You appear to be using OS version 2.6.35-24-generic.

WARNING: The build is being done on Linux Unknown linux.
        The official linux builds use Linux Advanced Server,
        specifically Linux Advanced Server release 2.1AS.
        The version found was '2.6.35-24-generic'.

WARNING: The directory HOTSPOT DOCS IMPORT PATH=/NO DOCS DIR
      does not exist, check your value of ALT HOTSPOT DOCS IMPORT PATH.

WARNING: The linux compiler must be version  3.2
      Specifically the GCC  compiler.
       You appear to be using compiler version: 4.4
      The compiler was obtained from the following location:
         /usr/bin/
      Please change your compiler.

WARNING: Importing CUPS from a system location

Sanity check passed.
make[1]: Leaving directory `/media/JDK6_build_area/jdk6u23/control/make'
```

(7)在设置好环境之后，接下来正式开始编译。在解压出来的源码包的 control/make 目录中连续执行两次下述命令即可。

```
$ make dev BUILD DEPLOY=false SKIP COMPARE IMAGES=true
ALT BOOTDIR=/usr/lib/jvm/java-6-openjdk ALT DEVTOOLS PATH=/usr/bin
HOTSPOT_BUILD_JOBS=2
```

其中后一次运行时间会比较长，大家需要耐心等待。之所以执行两次的原因是源码包中所包含的 Motif 不能一次顺利编译成功。当第一次执行上述命令后会看到如下类似错误：

```
gcc: /media/JDK6 build area/jdk6u23/control/build/linux-i586/motif-
i586/lib/libXm.a: No such file or directory
make[4]: *** [/media/JDK6 build area/jdk6u23/control/build/linux-
i586/lib/i386/motif21/libmawt.so] Error 1
make[4]: Leaving directory
`/media/JDK6 build area/jdk6u23/j2se/make/sun/motif21'
make[3]: *** [all] Error 1
make[3]: Leaving directory
`/media/JDK6 build area/jdk6u23/j2se/make/sun'
```

```
make[2]: *** [all] Error 1
make[2]: Leaving directory `/media/JDK6 build area/jdk6u23/j2se/make'
make[1]: *** [j2se-build] Error 2
make[1]: Leaving directory `/media/JDK6 build area/jdk6u23/control/make'
make: *** [dev-build] Error 2
```

只要第二次编译就可以解决上述错误，build 成功后可以看到如下类似信息：

```
generic debug build build finished: 11-01-16 03:11
make[2]: Leaving directory `/media/JDK6 build area/jdk6u23/control/make'
Control workspace build finished: 11-01-16 03:11
make[1]: Leaving directory `/media/JDK6_build_area/jdk6u23/control/make'
```

通过如下命令，可以进入 build 出来的 JDK 映像的 bin 目录中。

```
rednaxelafx@vbox:/media/JDK6 build area/jdk6u23/control/make$ cd ../buil
d/linux-i586/j2sdk-image/bin/
rednaxelafx@vbox:/media/JDK6 build area/jdk6u23/control/build/linux-
i586/j2sdk-image/bin$ ./java -version
java version "1.6.0-internal"
Java(TM) SE Runtime Environment (build 1.6.0-internal-
rednaxelafx 16 jan 2011 02 10-b00)
  Java HotSpot(TM) Server VM (build 19.0-b09, mixed mode)
rednaxelafx@vbox:/media/JDK6 build area/jdk6u23/control/build/linux-
i586/j2sdk-image/bin$ cd ../../j2sdk-debug-image/fastdebug/bin/
rednaxelafx@vbox:/media/JDK6 build area/jdk6u23/control/build/linux-
i586/j2sdk-debug-image/fastdebug/bin$ ./java -version
java version "1.6.0-internal-fastdebug"
Java(TM) SE Runtime Environment (build 1.6.0-internal-fastdebug-
rednaxelafx 16 jan 2011 02 38-b00)
Java HotSpot(TM) Server VM (build 19.0-b09-fastdebug, mixed mode)
```

这样整个编译过程就结束了，编译出来的 JDK 里的 Jconsole 可以正常运行了，如图 2-38 所示。

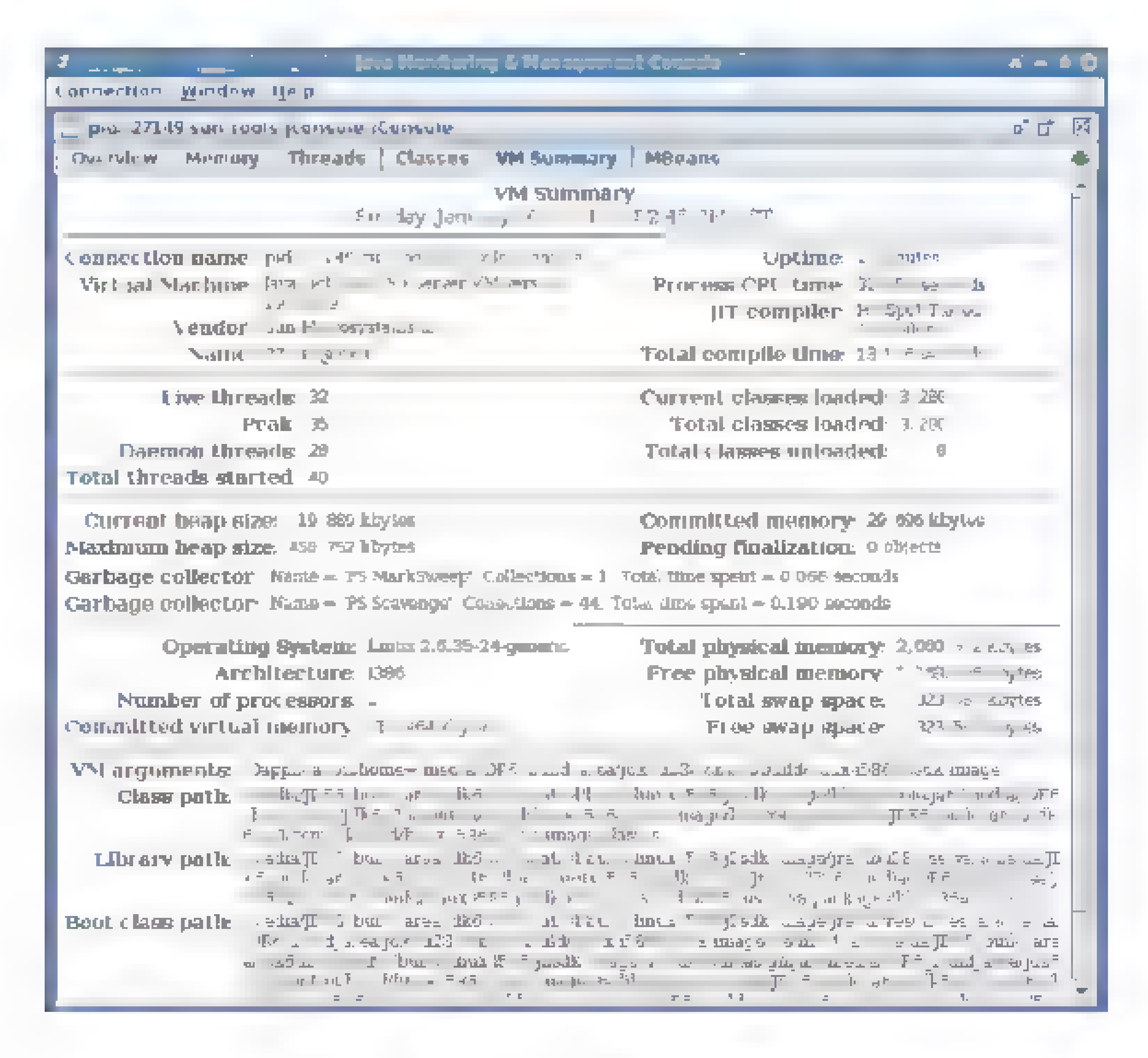


图 2-38　运行自己编译的 JDK

第 3 章

安全性的考虑

除了平台无关性以外，Java 还必须解决的另一个技术难题就是安全。因为网络中运行的多台计算机共享数据和分布式处理，所以它提供了一条侵入计算机系统的潜在途径，使得其他人有机会窃取、改变或破坏信息，盗取计算资源等。因此，将计算机联入网络引发了很多安全问题。为了解决由网络引起的安全问题，Java 体系结构采用了一个扩展的内置安全模型，这个模型随着 Java 平台的主要版本不断发展。

3.1 为什么需要安全性

Java 的安全模型是其多个重要结构特点之一，它能够使 Java 技术适应于网络环境的特殊性需求。因为很多网络应用是面向全体网民的，并且不分贵贱，所以这时候安全性就变得非常重要。如果在一个环境中，软件可以通过网络下载并在本地运行，这时就需要更加注意安全问题，例如 Java Applet 和 JINI 服务对象就是这样的例子。因为当用户在浏览器中打开网页时，会自动下载 applet 的 class 文件，用户很有可能会遇到来自不可靠来源的 applet。同样的道理，当一个 JINI 服务对象用 JINI 查找服务进行服务注册时，它的 class 文件将从服务供应商指定的代码库中下载。JINI 实现了一个自发的网络互联，在这个网络中，客户机进入一个新的环境查找并访问本地可用服务，因此，JINI 服务的客户机可能会遇到来自不可靠来源的服务对象。如果没有任何安全机制，这些代码自动下载的模式为恶意代码的发布提供了方便的途径。通过 Java 安全机制使 Java 适合于网络应用项目，为它们建立安全的网络移动代码提供了必要的可信机制。

Java 安全模型则主要用于保护终端用户免受从网络下载的，来自不可靠来源的、恶意程序的侵犯。为了达到这个目的，Java 提供了一个用户可配置的“沙箱”，在沙箱中可以放置不可靠的 Java 程序。沙箱对不可靠程序的活动进行了限制，程序可以在沙箱的安全边界内做任何事，但是不能进行任何跨越这些边界的举动。例如，原来在版本 1.0 中的沙箱对很多不可靠 Java Applet 的活动做了限制，主要包括：

- 对本地硬盘的读写操作。
- 进行任何网络连接，但不能连接到提供者 applet 的源主机。
- 创建新的进程。
- 装载新的动态链接库。

由于下载的代码不可能进行这些特定的操作，这使得 Java 安全模型可以保护终端用户避免受到有漏洞的代码的威胁。在沙箱内有严格的限制，其安全模型甚至规定了对不可靠代码能做什么、不能做什么，所以用户可以比较安全地运行不可靠代码。但是对于 1.0 系统的程序员和用户来说，这个最初的沙箱限制太过严格，善意的代码常常无法进行有效的工作。所以在后来的 1.1 版本中，对最初的沙箱模型进行了改进，引入了基于代码签名和认证的信任模式。签名和认证使得接收端系统可以确认一系列 class 文件已经由某一实体进行了数字签名(有效，可被信赖)。并且在经过签名处理以后，class 文件没有改动。这使得终端用户和系统管理员减少了对某些代码在沙箱中的限制，但这些代码必须已由可信任团体进行数字签名。

虽然 1.1 版本的安全 API 包含了对认证的支持，但是其实只是提供了完全信任和完全不信任策略。Java 1.2 提供的 API 可以帮助建立细粒度的安全策略，这种策略是建立在数字签名代码的认证基础上的。Java 安全模型的发展经历了 1.0 版本的基本沙箱，然后是 1.1 版本的代码签名和认证，最后是 1.2 版以后的细粒度访问控制。

3.2　沙箱模型的 4 种组件

在计算机世界，在个人电脑中运行一个软件的前提是必须信任它。普通用户只能通过小心地使用来自可信任来源的软件来达到安全性，并且定期扫描，检查病毒来确保安全性。一旦某个软件有权使用我们的系统，那么它将拥有对这台电脑的完全控制权。如果这个软件是恶意的，那么它就可以为所欲为。所以在传统的安全模式中，必须想办法防止恶意代码有权使用你的计算机。

3.2.1　沙箱模型介绍

沙箱安全模型使得工作变得容易，即使某个软件来自我们不能完全信任的地方，通过沙箱模型可以使我们接受来自任何来源的代码，而不是要求用户避免将来自不信任站点的代码下载到机器上。当运行来自不可靠来源的代码时，沙箱会限制它进行任何可能破坏系统的动作指令。并且在整个过程中，无需我们指出哪些代码可以信任，哪些代码不可以信任，也不必扫描查找病毒。沙箱本身限制了下载的任何病毒或其他恶意的，有漏洞的代码，使得它们不能对计算机进行破坏。

如果你还有疑问，在确信它能保护你之前，用户必需确认沙箱没有任何漏洞。为了保证沙箱没有漏洞，Java 安全模型对其体系结构的各方面都进行了考虑。如果在 Java 体系结构中有任何没有考虑到安全的区域，恶意的程序员很可能会利用这些区域来绕开沙箱。因此，为了对沙箱有一个了解，我们必须先看一下 Java 体系结构的几个不同部分，并且理解它们是怎样一起工作的。

下面是组成 Java 沙箱的基本组件。

- 类加载体系结构。
- class 文件检验器。
- 内置于 Java 虚拟机(及语言)的安全特性。
- 安全管理器及 Java API。

Java 的上述安全模型的前三个部分——类加载体系结构、class 文件检验器、Java 虚拟机(及语言)的安全特性一起达到一个共同的目的：保持 Java 虚拟机的实例和它正在运行的应用程序的内部完整性，使得它们不被下载的恶意代码或有漏洞的代码侵犯。相反，这个安全模型的第四个组成部分是安全管理器，它主要用于保护虚拟机的外部资源不被虚拟机内运行的恶意或有漏洞的代码侵犯。这个安全管理器是一个单独的对象，在运行的 Java 虚拟机中，它在对于外部资源的访问控制起中枢作用。

3.2.2　类加载体系结构

类加载器要加载一个类，它首先检查此类是否已被加载，然后再委托双亲加载器加载此类，它的双亲加载器再委托它的双亲，这样一直委托到启动加载器，启动加载器再从核心 API 查找此类，如果有就返回此类，否则它的子加载器就查找此类，如果都没有就抛出

ClassNotFound 的异常。

这种委托双亲模式的好处是：启动类加载器可以抢在标准扩展类装载器之前去装载类，而标准扩展类装载器可以抢在类路径加载器之前去装载那个类，类路径装载器又可以抢在自定义类加载器之前去加载它。所以 Java 虚拟机先从最可信的 Java 核心 API 查找类型，这是为了防止不可靠的类扮演被信任的类。假设网络上有一个名为 java.lang.Integer 的类，它是某个黑客为了想混进 java.lang 包所起的名字，实际上里面含有恶意代码，但是这种伎俩在双亲模式加载体系结构下是行不通的，因为网络类加载器在加载它的时候，首先调用双亲类加载器，这样一直向上委托，直到启动类加载器，而启动类加载 器在核心 Java API 里发现了这个名字的类，所以它就直接加载 Java 核心 API 的 java.lang.Integer 类，然后将这个类返回，所以自始至终网络上的 java.lang.Integer 的类是不会被加载的。

但是如果这个移动代码不是去试图替换一个被信任的类(就是前面说的那种情况)，而是想在一个被信任的包中插入一个全新的类型，情况会怎样呢？比如一个名为 java.lang.Virus 的类，经过双亲委托模式，最终类装载器试图从网络上下载这个类，因为网络类装载器的双亲们都没有这个类(当然没有了，因为是病毒吗)。假设成功下载了这个类，那你肯定会想，Virus 和 lang 下的其他类都保存在 java.lang 包下，暗示这个类是 Java API 的一部分，那么是不是也拥有修改 Java.lang 包中数据的权限呢？答案当然不是，因为要取得访问和修改 java.lang 包中的权限，java.lang.Virus 和 java.lang 下的其他类必须是属于同一个运行时包的。运行时包是指由同一个类装载器装载的、属于同一个包的、多个类型的集合。java.lang.Virus 和 java.lang 下的其他类不是同一个类装载器装载的，java.lang.Virus 是由网络类装载器装载的。

在 Java 沙箱中，类装载器体系结构是第一道防线。类装载器体系结构在以下三方面对 Java 的沙箱起作用。

(1) 防止恶意代码去干涉善意的代码；

(2) 守护了被信任的类库的边界；

(3) 将代码归入某类，该类确定了代码可以进行哪些操作。

类装载器体系结构可以防止恶意的代码去干涉善意的代码，这是通过为由不同的类装载器装入的类提供不同的命名空间来实现的。命名空间由一系列唯一的名称组成，每一个被装载的类有一个名字，这个命名空间是由 Java 虚拟机为每一个类装载器维护的。例如，一旦 Java 虚拟机将一个名为 Volcano 的类装入一个特定的命名空间，它就不能再装载名为 Volcanno 的其他类到相同的命名空间了。可以把多个 Volcanno 类装入一个 Java 虚拟机中，因为可以通过创建多个类装载器在一个 Java 应用程序中创建多个命名空间。如果在一个运行的 Java 应用程序中创建了三个独立的命名空间(为三个类装载器中的每一个都创建一个命名空间)，就可以通过为每个命名空间装载一个 Volcano 类，从而满足将三个不同的 Volcano 类装载到应用陈旭中的要求。

通过命名空间可以帮助安全的实现，因为我们可以有效地在装入不同命名空间的类之间设置一个防护罩。在 Java 虚拟机中，在同一个命名空间中的类可以直接进行交互。除非显式地提供允许它们进行交互的机制，否则在不同的命名空间中的类甚至不会察觉彼此的存在。一旦加载后，如果一个恶意的类被赋予权限访问其他虚拟机加载的当前类，它就可

以潜在地知道一些它不应该知道的信息，或者干扰程序的正常运行。

类装载器体系结构守护了被信任的类库的边界，这是通过分别使用不同的类装载器装载可靠的包和不可靠的包来实现的。虽然通过赋给成员受保护的访问限制，可以在同一个包中的类型间授予彼此访问的特殊权限，但这种特殊的权限只能授给在同一个包中的运行时成员，而且它们必须是由同一个类装载器装载的。

用户自定义类装载器经常依赖于其他类装载器，至少依赖于虚拟机启动时创建的启动类装载器，类可以帮助它实现一些类装载请求。在 1.2 版本以前，非启动类装载器必须显式地求助于其他类装载器，类装载器可以请求另一个用户自定义的类装载器来装载类，这个请求是通过对被请求的用户自定义类装载器调用 loadClass()来实现的。除此之外，类装载器也可以通过调用 findSystemClass()来请求启动类装载器来装载类，这是类 ClassLoader 中的一个静态方法。在 1.2 版本中，类装载器请求另一个类装载器来装载类型的过程被形式化，成为双亲委派模式。从 1.2 版本开始，除启动类装载器意外的每一个类装载器，都有一个“双亲”类装载器，在某个特定的类装载器试图以常用方式装载类型以前，它会先默认地将这个任务“委派”给它的双亲，目的是请求它的双亲来装载这个类型。这个双亲再一次请求它自己的双亲来装载这个类型。这个委派的过程一直向上继续，直到达到启动类装载器，通常启动类装载器是委派链中的最后一个类装载器。如果一个类装载器的双亲类装载器有能力来装载这个类型，则这个类装载器返回这个类型。否则这个类装载器试图自己来装载这个类型。

在 1.2 版本以前的大多数虚拟机中，内置的类装载器负责在本地装载可用的 class 文件。这些 class 文件通常包括如下元素：

- 要运行的 Java 应用程序的 class 文件。
- 此应用程序所需要的任何类库。
- 在这些类库中包含 JavaAPI 的基本 class 文件。

要想找到上述三类被请求类型的 class 文件，需要分析具体实现细节来实现。尽管如此麻烦，但是许多实现都是按照类路径(Classpath)指明的顺序查找目录和 Jar 文件的。

在 1.2 后面的版本中，装载本地可用的 class 文件的工作被分配到多个类装载器中，刚才称为原始类装载器的内置的类装载器被重新命名为启动类装载器，这表示它现在只负责装载那些核心 Java API 的 class 文件。因为核心 Java API 的 class 文件是用于“启动”Java 虚拟机的 class 文件，所以启动类装载器的名字也因此而得。

在 1.2 后面的版本中，通过用户自定义类装载器负责其他 class 文件的装载，例如实现应用程序运行的 class 文件，实现安装或下载标准扩展的 class 文件等。当 1.2 版本的 Java 虚拟机开始运行时，在应用程序启动以前，会至少创建一个用户自定义类装载器，很有可能会创建多个。所有这些类装载器被连接在一个“双亲 孩子”的关系链中，在这条链的顶端是启动类装载器，在这条链的末端是一个在 1.2 版本中被称为“系统类装载器”的类装载器。在 1.2 版本以前，“系统类装载器”这个名字有时指内置的类装载器，它也被称作原始类装载器。在 1.2 版本中，系统类装载器这个名字有了更正式的定义，它是指由 Java 应用程序创建的，新的用户定义类装载器的默认委派双亲。这个默认的委派双亲通常是一个用户自定义的类装载器，它装载应用程序的初始类，但是它也可能是任何用户自定

义的类装载器，这是由实现 Java 平台的设计者决定的。

例如一个 Java 应用程序装载了一个类装载器，这个装载器是通过网络下载来装载 class 文件的。假如我们在虚拟机上运行这个应用程序，在启动时实例化了如下两个用户自定义类装载器。

- 一个“已安装扩展”的类装载器。
- 一个“类路径”类装载器。

上述类装载器和启动类装载器一起连入一个双亲-孩子关系链中，如图 3-1 所示。类路径的类装载器的双亲是已安装了扩展的类装载器，而它的双亲是启动类装载器，在图 3-1 中，类路径装载器被设计成系统类装载器，新的用户自定义类装载器的默认委派双亲被应用程序实例化。假设当应用程序实例化它的网络类装载器时，它指明了系统类装载器作为它的双亲。

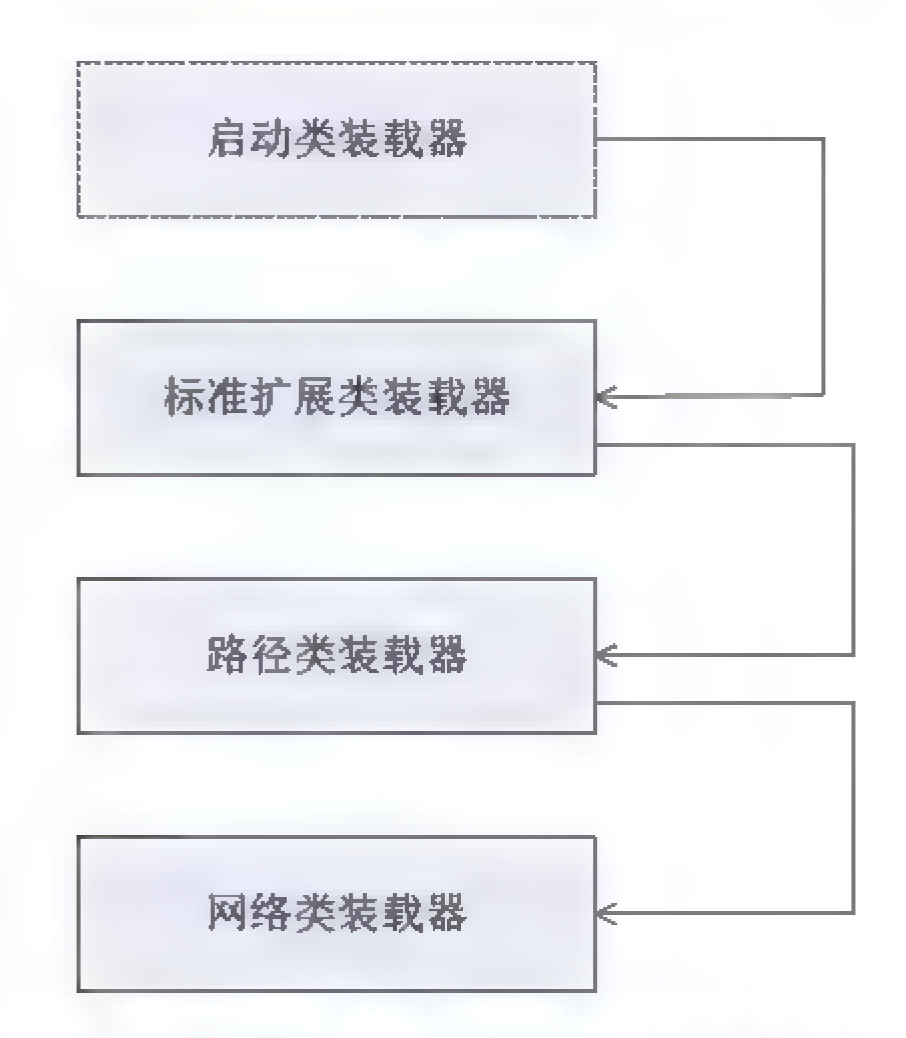


图 3-1 “双亲-孩子”类装载器委派链

假设在运行 Java 程序的过程中，类装载器发出一个装载 Volcano 类的请求，类装载器必须先询问它的双亲——类路径类装载器，然后查找并装载这个类。这个类路径类装载器依次将向它自己的双亲发出同样的请求，它的双亲即为安装扩展的类装载器。这个类装载器也是首先将这个请求委派给它自己的双亲——启动类装载器。假设 Volcano 类不是 Java API 的一部分，也不是一个已安装扩展的一部分，也不在类路径上，所有这些类装载器将返回而不会提供一个名为 Volcanno 的已装载类。当类路径装载器回答，它和所有它的双亲都不能装载这个类时，你的类装载器可能将试图用它自己特定的方式来装载 Volcano 类，它会通过网络下载 Volcano。假设你的类装载器可以下载类 Volcano，这样类 Volcano 就可以在应用程序以后的执行过程中发挥作用。假设在以后的某一时刻类 Volcano 的一个方法首次被调用，并且那个方法引用了 Java API 中的类 java.util.HashMap，因为这个引用是首次被运行的程序使用，所以虚拟机会请求你的类装载器(装载 Volcano 的类装载器)来装载 java.util.HashMap。就像以前一样，我们的类装载器首先将请求传递给它的双亲类装载器，然后这个请求一路委派，直到委派给启动类装载器。但是这次，启动类装载器可以将 java.util.HashMap 类返回给你的类装载器，因为启动类装载器可以找到这个类，所以已安

装扩展的类装载器就不必在已安装扩展中查找这个类型，类路径类装载器也不必在类路径中查找这个类型，同样我们的类装载器也不必从网上下载这个类。所有这些类装载器仅需要返回由启动类装载器返回的类 java.util.HashMap。从这时开始，无论类 Volcano 何时引用名为 java.util.HashMap 的类，虚拟机就可以直接使用这个 java.util.Hashmap 类了。

综上所述，类装载器的体系结构是通过剔除装作被信任的不可靠类来保护可信任类库的边界的。如果某个恶意的类可以成功地欺骗 Java 虚拟机，使 Java 虚拟机相信它是一个来自 Java API 的可信任类，那么这个恶意的类可能突破沙箱的阻隔。为了防止不可靠的类扮演被信任的类，类装载器的体系结构阻塞了危及 Java 运行时安全的潜在途径。

在有双亲委派模式的情况下，启动类装载器可以抢在标准扩展类装载器之前去装载类，而标桩扩展类装载器可以抢在类路径装载器之前去装载那个类，类路径装载器又可以抢在网络类装载器之前去装载它。这样，在使用双亲-孩子委派链的方法中，启动类装载器会在最可信的类库"核心 JavaAPI"中首先检查每个被装载的类型，然后依次到标准扩展，类路径上的本地类文件中检查。所以，如果网络类装载器装载的某段移动代码发出指示，试图通过网络下载一个和 JavaAPI 中某个类型同名的类型，例如 java.lang.Integer，它将不能取得成功。如果 java.lang.Interger 的 class 文件在 JavaAPI 中已经存在，它将被启动类装载器抢先装载，而网络类装载器将没有机会下载并定义名为 java.lang.Integer 的类，它只能使用由它的双亲返回的类，这个类应该是由启动类装载器装载的。用这种方法，类装载器的体系结构可以防止不可靠的代码用它们自己的版本替代可信任的类。

再次做一个假设，如果这个移动代码不是去试图替换一个被信任的类，而是想在一个被信任的包中插入一个全新的类型，情况会怎样呢？如果在刚才那个例子中，要求网络类装载器装载一个名为 java.lang.Virus 的类时。像以前一样，这个请求将首先被一路向上委派给启动类装载器，虽然这个启动类装载器负责装载核心 JavaAPI 的 class 文件，包括包 java.lang，但是它无法在包 java.lang 中找到名为 Virus 的成员。假设这个类在已安装扩展以及本地类路径中也找不到，我们的类装载器将试图从网络上下载这个类。

假设我们的类装载器成功下载并定义了名为 java.lang.Virus 的类，则 Java 允许在同一个包中的类拥有彼此访问的特殊权限，而在此包之外的类则没有这个权限。因为我们的类装载器装载了一个名为 java.lang.Virus 的类，则表示这个类是 Java API 的一部分，它会得到访问 java.lang 中被信任类的特殊访问权限，并且可以使用这个特殊的访问权限达到不可告人的目的。类装载机制可以防止这个代码得到访问 java.lang 包中被信任类的访问权限，因为 Java 虚拟机只把彼此访问的特殊权限授予由同一个类装载器装载到同一个包中的类型。因为 Java API 的包 java.lang 中被信任类是由启动类装载器装载的，而恶意的类 java.lang.Virus 是由网络类装载器装载的，所以这些类型不属于同一个运行时包。运行时包这个名词，是在 Java 虚拟机第 2 版规范中第一次出现的，它指由同一个类装载器装载的，属于同一个包的多个类型的集合。在允许两个类型之间对包内可见的成员进行访问前，虚拟机不但要确定这两个类型属于同一个包，还必须确认它们属于同一个运行时包——它们必须是由同一个类装载器装载的。这样因为 java.lang.Virus 和来自核心 JavaAPI 的 java。Lang 的成员不属于同一个运行时包，java.lang.Virus 就不能访问 JavaAPI 和 java.lang 保重的类型和包内可见的成员。

之所以提出运行时包的概念，目的之一是使用不同的类装载器装载不同的类。启动类装载器装载核心 JavaAPI 的 class 文件，这些 class 文件是最可信的。已安装扩展的类装载器装载来自于任何已安装扩展的 class 文件，已安装扩展是非常可信的，但这个可信度是在一定程度上的可信度，它们不能简单地通过将新类型插入到 JavaAPI 的包中来获得对包内可见成员的访问权，这是因为已安装扩展是由不同于核心 API 的类装载器装载的。同样，由类路径类装载器在类路径中发现的代码不能访问已安装扩展或 JavaAPI 中的包内可见成员。

类装载器可以使用另一种方法来保护被信任的类库边界，它只需通过简单地拒绝装载特定的禁止类型就可以了。例如，我们可能已经安装了一些包，在这些包中包含了应用程序需要装载的类，这些类必须是由网络类装载器的双亲“类路径装载器”来装载的，而不是由网络类装载器来装载的。假设已经创建了一个名为 absolutepower 的包，并且将它安装在本地类路径中的某个地方，在这里它可以被类路径类装载器访问到。而且假设你不想让由自己的类装载器装载的类，能装载来自 absolutepower 包中的任何类。在这种情况下，必须编写自己的类装载器，让它做的第一件事就是确认被请求的类不是 absolutepower 包中的一个成员。如果这样的类被请求装载，你的类装载器将抛出一个安全异常，而不是将这个类的名字传给双亲类装载器。

类装载器要知道一个类是否来源已一个被禁止的包，例如 absolutepower，唯一的方法是通过它的类名来检测。因为类名 absolutepower.FancyClassLoader 指明了它是包 absolutepower 的一部分，而包 absolutepower 被列在被禁止的包列表中，所以的类装载器应该立即抛出一个安全异常。

除了屏蔽不同命名空间中的类并保护被信任的类库的边界外，类装载器还有另外的安全作用。类装载器必须将每一个被装载的类放置在一个保护域中，一个保护域定义了这个代码在运行时将得到怎样的权限。这是类装载器的一个非常重要的安全工作。

3.2.3 class 文件检验器

class 文件检验器的功能是，保证类装载器装载的 class 文件内容有正确的内部结构，并且这些 class 文件相互间协调一致。如果 class 文件检验器在 class 文件中发现问题时会抛出异常。好的 Java 编译器不应该产生畸形的 class 文件，但是 Java 虚拟机并不知道某个特定的 class 文件是如何被创建的。因为一个 class 文件实质上是一个字节序列，所以虚拟机无法分辨特定的 class 文件是由正常的 Java 编译器产生的，还是由黑客打造的，因为黑客可能威胁虚拟机的完整性。所以，所有的 Java 虚拟机的实现必须有一个 class 文件检验器，文件检验器可以调用 class 文件以确保这些定义的类型可以安全的使用。

健壮性是 class 文件检验器实现的安全目标之一。如果某个有漏洞的编译器或比较牛黑客产生了一个 class 文件，而在这个 class 文件中包含了一个方法，在这个方法的字节码中包含了一条跳转到方法之外的指令，那么一旦调用这个方法，则会导致虚拟机崩溃的结果。正因为如此，所以出于健壮性的考虑，迫切需要虚拟机检验它装载的字节码是否完整。

Java 虚拟机的 class 文件检验器在字节码执行之前，必须完成大部分检验工作。它只

在执行前而不是在执行中对字节码进行一次分析(并验证它的完整性)，每一次遇到一个跳转指令时都进行检验。作为字节码确认工作的一部分，虚拟机将确认所有的跳转指令会到达另一条合法的指令，而且这条指令是在这个方法的字节码流中。在大多数情况下，在执行前就对所有字节码进行一次检查，对于保证健壮性来说就够了，而不必在它运行时每次都检验每一条字节码指令。

class 文件检验器要进行如下四趟独立的扫描来完成它的操作。

- 第一趟扫描时在类被装载时进行的，在这次扫描中，class 文件检验器检查这个 class 文件的内部结构，以保证它可以被安全地编译。
- 第二和第三趟扫描时在连接过程中进行的，在这两次扫描中，class 文件检验器确认类型数据遵从 Java 编程语言的定义，包括检验它所包含的所有字节码的完整性。
- 第四趟扫描是在进行动态链接的过程中解析符号引用时进行的，在这次扫描中，class 文件检验器确认被引用的类、字段以及方法确实存在。

在接下来的内容中，将对这四趟扫描的具体过程进行详细讲解。

1) 第一趟：class 文件的结构检查

在第一趟扫描中，对每一段将被当作类型导入的字节序列，class 文件检验器都会确认它是否符合 Javaclass 文件的基本结构。在这次扫描中，检验器将进行许多检查，例如每个 class 文件必须以四个同样的字节开始：魔数 0xCAFEBABE。这个魔数的功能是让 class 文件分析器很容易分辨出某个文件有明显问题而加以拒绝，此文件可能是破坏了的 class 文件，或者根本就不是 class 文件。这样，class 文件检验器所做的第一件事可能是检查导入的文件是否是以魔数开头。检验器还必须确认在 class 文件中声明的主版本号和次版本号，这个版本号必须在这个 Java 逊尼基实现可以支持的范围之内。

在第一趟扫描中，class 文件检验器必须检验确认这个 class 文件没有被删节，尾部也没有附带其他的字节。虽然不同的 class 文件有不同的长度，但是 class 文件中包含的每一个组成部分都声明了它的长度和类型。检验器可以使用组成部分的类型和长度来确定整个 class 文件的正确的总长度。用这种方法，它就可以检查一个装入的文件，其长度是否和它里面的内容相一致。

第一趟扫描的主要目的就是保证这个字节序列正确地定义了一个新类型，它必须遵从 Java 的 class 文件的固定格式，这样它才能被编译成在方法区中的内部数据结构。第二、第三和第四趟扫描不是在符合 class 文件格式的二进制数据上进行的，而是在方法区中的，由实现决定的数据结构上进行的。

2) 第二趟：类型数据的语义检查

在第二趟扫描中，在检查时 class 文件检验器不需要查看字节码，也不需要查看和装载任何其他类型。在这趟扫描中，检验器会查看每个组成部分，确认它们是否是其所属类型的实例，它们的结构是否正确。例如方法描述符(它的返回类型，以及参数的类型和个数)在 class 文件中被存储为一个字符串，这个字符串必须符合特定的上下文无关文法。检验器对每个组成部分进行检查的目的之一是，为了确认每个方法描述符都是符合特定语法的、格式正确的字符串。

除此之外，class 文件检验器检查这个类本身是否符合特定的条件，它们是由 Java 编程语言规定的。例如，检验器强制规定除 Object 类以外的所有类，都必须有一个超类。在第二趟扫描中，检验器还要检查 final(最终的)类没有被子类化，而且 final 方法没有被覆盖。还要检查常量池中的条目是合法的，而且常量池的所有索引必须指向正确类型的常量池条目。也就是说，class 文件检验器在运行时检查了一些 Java 语言应该在编译时遵守的强制规则。因为检验器并不确定 class 文件是否是由一个善意的、没有漏洞的编译器产生的，所以它会检查每个 class 文件，以确保这些规则得到遵守。

3) 第三趟：字节码验证

在 class 文件检验器成功地进行了前面的两趟检查之后，接下来开始把注意力放在字节码上，所以第三趟扫描被称为“字节码验证”。在这趟扫描中，Java 虚拟机对字节流进行数据流分析，这些字节流代表的是类的方法。为了理解字节码检验器，必须对字节码和栈帧有一定的了解。

字节码流代表了 Java 的方法，它是由被称为操作码的单字节指令组成的序列，每一个操作码后都跟着一个或多个操作数。操作数用于在 Java 虚拟机执行操作码指令时提供所需的额外的数据。执行字节码时，依次执行每个操作码，这就在 Java 虚拟机内构成了执行的线程。每一个线程被授予自己的 Java 栈，这个栈是由不同的栈帧构成的。每一个方法调用将获得一个自己的栈帧——栈帧其实就是一个内存片段，其中存储着局部变量和计算的中间结果。在栈帧中，用于存储方法的中间结果的部分被称为该方法的操作数栈。操作码和它的(可选的)操作数可能指存储在操作数栈中的数据，或存储在方法栈帧中局部变量中的数据。这样，在执行一个操作码时，除了可以使用紧随其后的操作数，虚拟机还可以使用操作数栈中的数据，或局部变量中的数据，或是两者都用。

字节码检验器需要进行大量的检查，目的是确保采用任何路径在字节码流中都得到一个确定的操作码，确保操作数栈总是包含正确的数值以及正确的类型。它必须保证局部变量在赋予合适的值以前不能被访问，而且类的字段中必须总是被赋予正确类型的值，类的方法被调用时总是传递正确数值和类型的参数。字节码检验器还必须保证每个操作码都是合法的，即每一个操作码都有合法的操作数，以及对每一个操作码，合适类型的数值位于局部变量中或是在操作数栈中。这些仅仅是字节码检验器所做的大量检验工作中的一小部分，在整个检验过程通过后，它就能保证这个字节码流可以被 Java 虚拟机安全地执行。

字节码检验器并不试图检测出所有的安全问题。如果要这样的话，它将会遇到“停机问题”。停机问题是计算机科学领域的一个著名论题：即不可能写出一个程序，用它来判断作为其输入而读入的某个或层序在执行时是否停机。一个程序是否会停机被称为是程序的“不可判定”特性，因为不可能写出一个程序，让它 100%地告诉你任何一个给定的程序是否含有这种特性。停机问题的不可判定性可以扩展成计算机程序的许多特性，如一个 Java 字节码的集合是否能被虚拟机安全地执行。如果不是，那么，这些字节码可能可以被虚拟机安全地执行，也可能不能安全地执行。这样，通过识别一些安全的字节码流，但不是全部，检验器就绕过了停机问题。由于字节码检验器强制检查的特性，只要定义好规则，任何程序只要可以用 Java 编程语言书写，编译器就可以确保编译出来的字节码可以被检验器通过。有些程序虽然不能用 Java 编程语言源代码表达出来，但仍可以通过检验器的

检验。另外还有些程序，它们实际上也能被 Java 虚拟机安全地执行，却不能通过检验器的检验。

在第一、第二和第三趟扫描中，class 文件检验器能够保证导入的 class 文件构成合理，符合 Java 编程语言的限制条件，并且包含的字节码可以被 Java 虚拟机安全地执行。如果 class 文件检验器发现其中任何一点不正确，都会抛出一个错误，这个 class 文件将不会被程序使用。

4) 第四趟：符号引用的验证

在动态链接的过程中，如果包含在一个 class 文件中的符号引用被解析时，class 文件检验器将进行第四趟检查。在第四趟检查中，Java 虚拟机将追踪那些引用——从被验证的 class 文件到被引用的 class 文件，以确保这个引用是正确的。因为第四趟扫描必须检查被检测的 class 文件以外的其他类，所以这次扫描可能需要装载新的类。大多数 Java 虚拟机的实现采用延迟装载类的策略，直到类真正地被程序使用时才装载。即使一个实现确实预先装载了这个类，这是为了加快装载过程的速度，那它还是会表现为延迟加载。例如，如果 Java 虚拟机在预先装载中发现它不能找到某个特定的被引用的类，它并不在当时抛出 NoClassDefFoundError 错误，而是直到(或者除非)这个被引用类首次被运行程序使用时才抛出。这样，如果 Java 虚拟机进行预先链接，第四趟扫描可以紧随第三趟扫描发生。但是如果 Java 虚拟机在某个符号引用第一次被使用时才进行解析，那么第四趟扫描将在第三趟扫描以后很久、当字节码被执行时才进行。

class 文件的检验器的第四趟扫描仅仅是动态链接过程的一部分。当一个 class 文件被装载时，它包含了对其他类的符号引用以及它们的字段和方法。一个符号引用是一个字符串，它给出了名字，并且可能还包含了其他关于这个被引用项的信息——这些信息必须足以唯一地识别一个类字段或方法。这样，对于其他类的符号引用必须给出这个类的全名；对于其他类的字段的符号引用必须给出类名、字段名以及字段描述符；对于其他类中的方法的引用必须给出类名、方法名以及方法的描述符。

动态链接是一个将符号引用解析为直接引用的过程。当 Java 虚拟机执行字节码时，如果它遇到一个操作码，这个操作码第一次使用一个指向另一个类的符号引用，那么虚拟机就必须解析这个符号引用。虚拟机在解析时需要执行下面的三个基本任务：

- 查找被引用的类(如果必要的话就装载它)。
- 将符号引用替换为直接引用，这样当它以后再次遇到相同的引用时，它就可以立即使用这个直接引用，而不必花时间再次解析这个符号引用了。
- 当 Java 虚拟机解析一个符号引用时，class 文件检验器的第四趟扫描确保了这个引用是合法的。当这个引用是个非法引用时——例如，这个类不能被装载，或这个类的确存在，但是不包含被引用的字段或方法——class 文件检验器将抛出一个错误。

继续以类 Volcano 为例进行讲解。如果类 Volcano 中的某个方法调用了类 Lava 中的某个方法，此 Lava 类中的方法的全名和描述符将包含在 Volcano 的 class 文件的二进制数据中。当 Volcano 类的方法在执行过程中第一次调用 Lava 的方法时，Java 虚拟机必须确认在类 Lava 中存在这个方法，并且这个方法的名字和描述符与类 Volcano 中期待的相匹配。如

果这个符号引用是正确的，那么虚拟机将把它替换为一个直接引用，例如一个指针，从那时开始将使用这个指针。如果类 Volcano 中的符号引用不能匹配 Lava 类中的任何方法时，第四趟扫描验证失败，Java 虚拟机将抛出一个 NoSuchMethodError。

正因为 Java 程序是动态链接的，所以 class 文件检验器在第四趟扫描中，必须检查相互引用的类之间是否兼容。如果我们修改了一个类，Java 编译器常会重编译这些类，从而在编译时检测是否有任何的不兼容性。但是也有很多时候，编译器不对受影响的类进行重编译，例如，如果正在开发一个大型系统，很可能将系统分割成几个部分放入包中。如果对每个包进行独立的编译，当改动包中的一个类时，可能将导致同一个包内受影响的那些类需要重新编译，但是对于其他包来说就不需要进行重编译了。此外，如果使用了其他人的一些包，尤其是程序在运行时通过网络下载了一些类，就不可能在编译时检验兼容性。这就是为什么 class 文件检验器在第四趟扫描时，必须在运行时检查兼容性的原因。

接下来举一个修改不兼容的例子。假设用一个 Java 编译器编译了类 Volcano，因为类 Volcano 中的一个方法调用了另一个类 Lava 中的方法，Java 编译器将查找类 Lava 的 class 文件或源文件，以确认类 Lava 中是否有一个方法具有这个名字，返回类型和相同数量并且类型相同的参数。如果编译器不能找到任何名为 Lava 的类，或者找到一个 Lava 类，但是这个类中不包含想要找的方法，那么编译器将产生一个错误，并且将不为 Volcano 类生成 class 文件。否则，Java 编译器可以为 Lava 类生成与 Volcano 类不兼容的 class 文件。如果 Lava 类不引用 Volcano 类，当改变 Lava 类中被 Volcano 引用的方法的方法名时，这样只会对 Lava 类进行重编译，如果在运行你的程序时，你试图使用新版本的 Lava，而继续使用老版本的 Volcano 类，而这个 Volcano 类将和新版本的 Lava 不兼容，那么当 Volcano 试图调用 Lava 中不再存在的方法时，在第四趟 class 文件检验中将抛出一个 NoSuchMethodError。

在这种情况下，对于类 Lava 的修改将打破它与已有的 Volcano 的 class 文件的二进制兼容性。实际上，当更新一个已经在使用的类库，并且它的新版本与已有的代码不兼容时，这种情况就会发生。为了能方便地修改类库的代码，Java 编程语言被设计成允许对一个类做多种修改，但并不要求对依赖于它的那些类进行重编译。Java 语言规范中列出了用户可以做的多种改动，这些改动成为二进制兼容性规则。这些规则明确地定义了：在一个类中，哪些可以被修改、增加和删除，而并不破坏这个被修改的类与依赖于它的那些事先已存在的类之间的二进制兼容性。例如，像一个类中增加一个新的方法始终是一个影响二进制兼容性的改动，但是不能删除一个正在被其他类使用的方法。所以在这种情况下，当改变了 Lava 类中被 Volcano 调用的方法的名称时，就破坏了二进制兼容规则，因为实际上是删除了一个老的方法，并加入了一个新的方法。但是如果你加入了一个新的方法，并改写了老的方法，让它调用新的方法，那么这个改动和所有早已使用的 Lava 的、事先已有的 class 文件是二进制兼容的，包括 Volcano。

3.2.4　内置于 Java 虚拟机(及语言)的安全特性

在 Java 的虚拟机中会装载一个类，当对此类进行了从第一到第三趟的 class 文件检验处理之后，就可以运行这些字节码了。除了对符号引用的检验(class 文件检验的第四趟扫

描)，Java 虚拟机在执行字节码时还需要进行其他一些内置的安全机制的操作。这些机制大多数是 Java 的类型安全的基础，它们作为 Java 编程语言保证 Java 程序健壮性的特性，也是 Java 虚拟机的特性。下面就是内置于 Java 虚拟机(及语言)的安全特性。

- 类型安全的引用转换。
- 结构化的内存访问(无指针算法)。
- 自动垃圾收集(不必显示地释放被分配的内存)。
- 数组边界检查。
- 空引用检查。

保证一个 Java 程序只能使用类型安全的、结构化的方法去访问内存，Java 虚拟机使得 Java 程序更为健壮，也使得它们的运行更为安全。如果一个程序破坏内存、崩溃，或者可能导致其他程序崩溃，那么，他就是一个安全裂口。例如，如果正在运行一个任务关键的服务器进程，那么保证这个进程不能崩溃就非常重要。这种层次的健壮性在嵌入式系统中也非常重要，例如，蜂窝电话，因为人们通常不希望重启机器。如果对内存的访问不加限制条件，会导致安全隐患的另一个原因是，一个老谋深算的黑客可能暗中利用它破坏安全系统。例如一个黑客知道一个类装载器在内存中的位置，他可以赋一个指针指向那块内存，从而对类装载器的数据进行操作。通过强制对内存的结构化访问，Java 虚拟机可以产生健壮的程序，而且还可以阻挠那些黑客，使他们不能为了达到某些目的而破坏 Java 虚拟机的内在存储。

内置在 Java 虚拟机中的另一个安全特性是作为内存的结构化访问的一个后备，也就是并未指明运行时数据空间在 Java 虚拟机内部是怎样分布的。运行时数据空间是指一些内存空间，Java 虚拟机用这些空间来存储运行一个 Java 程序时所需要的数据：Java 栈(每个线程一个)、一个存储字节码的方法区，以及一个垃圾收集堆(它用来存储由运行的程序创建的对象)。如果查看一个 class 文件的内部，将找不到任何内存地址。当 Java 虚拟机装载一个 class 文件时，由它决定将这些字节码以及其他从 class 文件中解析得到的数据放置在内存的什么地方。当 Java 虚拟机启动一个线程时，由它决定将它为这线程创建 Java 栈放到哪里，当它创建一个新的对象时，也是由它决定将这个对象放到内存中的什么地方。这样一个黑客就不可能仅凭 class 文件中的内容，就知道在内存中的哪些数据代表了这个类，或从那些这个类实例化而得到的对象(对于黑客来说)。更糟糕的是，黑客不能通过阅读 Java 虚拟机的规范来得到关于内存布局的任何信息。在规范中，并未说明 Java 虚拟机是怎样布局它的内存数据的。对于每个 Java 虚拟机的实现来说，由它的设计者来决定使用什么数据结构来表示运行时数据空间，并且将它们存放在内存的哪个位置。因此，即使黑客可以在一定程度上突破 Java 虚拟机的内存访问限制，他们也会在四处查找某些东西想进行暗中破坏时遇到困难。

通常来说，虚拟机会禁止对内存进行非结构化访问。其实 Java 虚拟机并不是必须主动强制正在运行的程序，因为这种禁止其实是字节码指令集本身的内在本质。就像在 Java 编程语言中，无法表达一个非结构化的内存访问那样，并且在字节码中也没有办法表达非结构化的内存访问，即使是我们自己编写的字节码。所以在禁止对内存的非结构化访问时，是对防止对内存恶意破坏的一种阻碍。

其实可以有法突破由支持 Java 虚拟机的类型安全机制所建立的安全屏障。虽然字节码指令集没有向用户提供不安全的，非结构化的内存访问方法，但是完全可以绕过字节码，即调用本地方法。当调用本地方法时，Java 安全沙箱会完全不起作用。首先，健壮性的保证对于本地方法并不适用，虽然不能通过 Java 方法破坏内存，但是可以通过本地方法来实现这个目的。并且最重要的是，本地方法没有经过 Java API，所以当一个本地方法试图进行破坏性动作时，安全管理器并没有被检查。这样，一旦某个线程进入一个本地方法，无论 Java 虚拟机内置了何种安全策略，只要这个线程运行这个本地方法，此时的安全策略将不再对这个线程适用。因为在调用本地方法时动态链接库是必需的，所以专门在安全管理器中包含了一个对应的处理方法，通过此方法可以确定一个程序是否能动态装载连接库。例如，因为不可靠的 applet 不能装载新的动态链接库，所以它们不能安装自己的新的本地方法，但是他们可以调用 Java API 中的方法，这些方法可能是本地的，但是这些方法是可信的。当线程调用本地方法时，这个线程就跳出了沙箱。所以对于本地方法来说，这个安全模型就和保障计算机安全的传统的安全模型完全相同，也就是说在调用本地方法前必须确认它是可信任的。

为了保证安全而内置于 Java 虚拟机的最后一个机制，就是异常的结构化错误处理。因为 Java 虚拟机支持异常，所以当一些潍坊安全的行为发生时会进行结构化处理，Java 虚拟机将抛出一个异常或者错误，而不会发生崩溃现象。这个异常或者错误会使这个错误线程的死亡，而不会使整个系统崩溃。抛出一个错误(和抛出一个异常对应)总是导致抛出错误的这个线程死亡。这对一个运行的 Java 程序来说通常是一个不便因素，但它不会导致整个程序的中止。如果这个程序还有一些线程正在正常工作，则这些线程有可能继续正常工作，即使它的同伴已经死亡。而抛出一个异常可能导致这个线程的死亡，但是它经常作为一种手段被使用，使程序能够将控制从发生异常的地方转到处理异常的情况。

3.2.5 安全管理器和 Java API

这是安全沙箱中，离我们程序员最接近的一环。SecurityMananger 安全管理器是一个 API 级别的、可以自定义的安全策略管理器，它深入到 Java API 中，在各处都可以见到它的身影。比如 SecurityClassLoader。

在默认情况下，Java 应用程序是不设置 SecurityManager 实例的，这意味着不会起到安全检查，这个实例需要我们在程序启动时通过 System.setSecurityManager 来设置。

在一般情况下，通过 SecurityManager.checkPermission(Permission perm)来完成权限检查。外部程序通过创建 Permission 实例传递给前面的检查。

Permission 是一个抽象类，需要继承它实现不同的权限验证，比如下面的 FilePermission 代表对某个文件的读写权限。

```
new FilePermission("test.txt", "read")
```

综上所述，Java 的沙箱安全模型最重要的优点之一是，这些组件中的类装载器和安全管理器是可以由用户定制的。通过定制这些组件，可以为 Java 程序创建个性化的安全策略。但是这种可定制性是需要代价的，因为这种体系结构的灵活性也对它本身产生一定的

风险。类装载器和安全管理器非常复杂，因此，单纯的定制操作也可能潜在地产生错误，从而开启安全漏洞。

在 Java API 的每一个主要版本中都进行了一些改进，使得创建定制的安全策略时出错机会更少。最重要的改变是是从 1.2 版本开始的，从此引入了一个新的且更为精细的访问控制体系结构。在以前的 1.0 版本和 1.1 版本中，访问控制包括安全策略规范和运行时安全策略的实施，它是由称作安全管理器的对象负责的。要在 1.0 版本和 1.1 版本中建立定制的策略，必须编写自己的定制安全管理器。在 1.2 版本中，可以利用 Java 2 平台提供的安全管理器，这个预先制作好的安全管理器，允许用户在一个和程序分离的 ASCII 策略文件中说明安全策略。在运行时，这个预先定制好的安全管理器获得一个类的帮助，来执行在策略文件中说明的安全策略。在版本 1.2 中引入的访问控制基础架构提供了灵活易定制的安全管理器的默认实现，这个默认实现可以满足大多数的安全需求。为了向后兼容，也为了使一些有特殊安全需要的团体可以对预先制作好的安全管理器中的默认功能进行改写，版本 1.2 的应用程序还可以安装它们自己的安全管理器。是使用预先制作的安全管理器，还是使用在它之上进行扩展的访问控制基础架构，用户使可以选择的。

前面介绍的三个 Java 安全模型共同实现了一个目的：保持 Java 虚拟机的实例和它正在运行的应用程序的内部完整性，使得它们不被下载的恶意或有漏洞的代码侵犯。这个安全模型的第四个组成部分是安全管理器，它主要用于保护虚拟机的外部资源不被虚拟机内运行的恶意或有漏洞的代码侵犯。这个安全管理器是一个单独的对象，在运行的 Java 虚拟机中，它在外部资源的访问控制中起到了中枢作用。

安全管理器定义了沙箱的外部边界。因为它是可定制的，所以它允许为程序建立自定义的安全策略。当 Java API 进行任何可能不安全的操作时，它都会向安全管理器请求许可，从而强制执行自定义的安全策略。要向安全管理器请求许可，JavaAPI 将调用安全管理器的“check”方法。因为这些方法的名都以“check”开头，所以他们被称为“check”方法。例如，安全管理器的方法 checkRead()决定了线程是否可以读取一个特定的文件，方法 checkWrite()决定了线程是否对一个特定的文件进行写操作。这些方法的实现定义了应用程序的定制安全策略。

因为 Java API 在进行一个可能不安全的操作前，总是检查安全管理器，所以 Java API 不会在安全管理器建立的安全策略之下执行被禁止的操作。如果安全管理器禁止这个操作，Java API 就不会执行这个操作。

当启动 Java 应用程序时还没有安全管理器，但是应用程序通过将一个指向 java.lang.SecurityManager 或是其子类的实例传给 setSecurityManager()，以此来安装安全管理器，这个动作是可选的。如果应用程序没有安装安全管理器，那么它将不会对请求 Java API 的任何动作做限制。Java API 将做任何被请求的事。如果应用程序确实安装了安全管理器，那么安全管理器将负责应用程序整个剩余的生命周期，它不能被替代、扩展或修改。从这一点开始，Java API 将只执行那些被安全管理器同意的请求。然而在 1.2 版本中，当前安装的安全管理器可以被允许替换它的代码所替换，这是通过对指向另一个不同的安全管理器对象的引用调用 System.setSecurityManager()实现的。

如果禁止了一个手检查的动作，则安全管理器的“check”方法将抛出一个安全异常，

如果这个动作被允许则简单地返回。所以当一个 Java API 即将进行一个潜在不安全的动作时，它将遵循以下两个步骤。

(1) Java API 的代码检查有没有安装安全管理器，如果没有安装，则跳过第二步直接继续这个潜在不安全的动作。

(2) 否则在第二步中将调用安全管理器中的合适的“check”方法。如果这个操作被禁止，那么此“check”方法会抛出一个安全异常，这将导致该 Java API 方法立即中止，这个潜在不安全的操作将不会被执行。相反，如果这个操作被允许，那么这个“check”方法将简单地返回。此时这个 Java API 方法将继续运行，并执行这个潜在不安全的操作。

3.3 浅谈安全管理器的必要性

在整个 Java 虚拟机系统中，安全管理器主要负责如下两个工作：

- 说明一个安全策略。
- 执行这个安全策略。

安全策略指明了哪种代码将被允许执行哪种操作，它是由安全管理器中的方法“check”的代码定义的。当它们被调用时，这个策略将被方法“check”所实施。在本节的内容中，将详细讲解安全管理器在虚拟机中的必要性。

3.3.1 公正评论安全管理器优点和弱点

客观公正地说，虽然安全管理器的可配置性是 Java 安全模型的最大优点之一，但是也存在一个潜在的弱点。例如编写一个安全管理器是一项复杂的任务，并且很可能会导致错误发生。在实现安全管理器的 check 方法时，任何错误都将变成运行时的安全漏洞。为了帮助开发人员和终端用户方便地，尽量正确地建立一个基于签名代码的细粒度的安全策略，例如类 java.lang.SecurityManger 是一个具体的类，此类提供了一个默认的安全管理器的实现。用户的应用程序可以显式地实例化并安装这个安全管理器，或者也可以让它自动安装。例如，在 JDK 1.7 中，可以在命令行使用 Djava.security.manager 选项来指明安装具体安全管理器。

具体安全管理器类允许用户不用 Java 代码定义自己的定制策略，而是使用一个称为策略文件的 ASCII 文件。在策略文件中，可以给代码来源授予权限。权限是用类定义的，它是 java.security.Permission 的子类。例如，java.io.FilePermission 表示了对一个文件的读写执行或者删除权限。代码来源是由代码库的 URL 和一些签名组成的，从这个 URL 可以装载代码，而签名则为这个代码作担保。当创建安全管理器时，它对策略文件进行解析，并创建 CodeSource 和 Permission 对象，这些对象被封装在一个单独的 Policy 对象中，此 Policy 对象代表了运行时的策略。任何时刻只能有一个 Policy 对象被封装。

类装载器将类型放到保护域中，保护域封装了授予代码来源的所有权限，这些代码来源由装载的类型代表。每一个被装载虚拟机中的类型都属于一个且只属于一个保护域。这个保护域会被记下来，并且在决定这个代码是否被允许执行一些可能不安全的操作时使用它。

3.3.2　方法 check

当调用具体安全管理器的方法 check 时，他们中的大多数都将请求传递给一个称为 AccessController 的类。此 AccessController 类使用了包含在保护域对象中的信息，这个对象所属的类的方法在调用栈中，AccessController 进行栈检查以确定这个操作能否被执行。

在古老的版本 1.0 和 1.1 中，每一个 check 方法都在它们的方法名中指出了将检查什么。为了检查能否读取一个特定的文件，JavaAPI 调用了安全管理器的 checkRead()方法，并且将被读取文件的路径名作为参数传入。例如在试图读取文件/tmp/finances.dat 之前，安全管理器调用了 checkRead(“/tmp/finances.dat”)。

安全管理器声明了 28 个 check 方法，下面详细列出了其中最为常用的 27 个 check 方法，同时也列出了它们被 Java API 代码触发调用时的潜在不安全动作：

- checkConnect(String host,int port)：功能是打开一个指定主机和端口号的 socket 连接前辈调用。
- checkConnect(String host, int port, Object context)：功能是在被传递的安全上下文中打开一个指定主机和端口号的 socket 连接前调用。
- checkAccept(String host, int port)：功能是接收一个来自于指定主机和端口号的 socket 连接前被调用。
- checkCreateClassloader()：功能是创建一个新的类加载器前被调用。
- checkAccess(Thead t)：功能是改变一个线程(如改变它的优先级、中止它等)前被调用。
- checkAccess(ThreadGroup t)：功能是改变一个线程组(如加入一个新的线程、设置守护进程等)前被调用。
- checkExists()：功能是应用程序退出前被调用。
- checkLink()：功能是装载一个包含本地方的动态库前调用。
- checkRead(FileDescription fd)：功能是读取指定的文件前被调用。
- checkRead(String file)：功能是读取指定的文件前被调用。
- checkRead(String file，Objectcontext)：功能是在被传递的安全上下文中读取指定的文件前被调用。
- checkWrite(FileDescription fd)：功能是对指定的文件进行写操作前被调用。
- checkWrite(Stirng file)：功能同上。
- checkListen(intport)：功能是在指定的本地端口号上等待连接前被调用。
- checkMulticase(IndeAddress maddr)：功能是在加入、离开、发送或者接收 IP 组播前被调用。
- checkMulticase(InedAddress maddr，byte t)：功能是在加入、离开、发送或者接收 IP 组播前被调用。此函数需要传递数据的大小作为参数。
- checkPropertiesAccess()：功能是访问和修改一般的系统属性前被调用。
- checkPropertiesAccess(String key)：功能是访问或修改指定的系统属性前被调用。
- checkTopLevelWindow(Object window)：功能是不出示任何警告地显示指定的窗

口前被调用。

- chceckPrintabAccess()：功能是初始化一个打印任务请求前被调用。
- checkSystemClipboardAccess()：功能是访问系统剪贴板前被调用。
- checkPackageAccess(String pkg)：功能是访问指定的包(被类装载器使用)中的类型前被调用。
- checkPackageDefinition(Stirng pkg)：功能是在指定的包(被类装载器使用)中加入一个新类前被调用。
- checkSetFactory()：功能是设置被 ServerSocket 或者 Socket 使用的 socket 类或者设置被 URL 使用的 URL 流处理器前被调用。
- checkMembetAccess()：功能是通过映像 API 访问类信息前被调用。

在后来的新版本中定义了一些许可类，这些类的实例代表了这样的代码，即它的操作是被允许的。例如在 1.2 版本的类 java.lang.SecurityManager 中添加了一对新的 check 方法，方法名都是 checkPermission()：

- checkPermission(Permission perm)：功能是进行某个操作(它需要指定的权限)前被调用。
- checkPermission(Permission perm，Objectcontext)：功能是在被传递的安全上下文中进行某个操作(它需要指定的权限)前被调用。

上述 checkPermission()方法能够接受一个 Permission 对象的引用，它指出了被请求的操作。这样此方法就提供了另一种方式，询问安全管理器是否可以执行一个潜在不安全的动作。例如要确定是否可以读文件/tmp/finances.dat，新版本中的 Java API 可以在两种方法中任选一种。Java API 可以采用老式的步骤，调用老式的方法 checkRead()，并将字符串/tmp/finance.dat 作为参数传递给它。或者 Java API 也可以采用新的方法，创建一个 java.io.FilePermission 对象，将字符串“/tmp/finance.dat”和“read”传给构造器 FilePermission，然后 Java API 将这个 Permission 对象传给安全管理器的方法 checkPermission()。

无论是使用老式方式调用一个老式的 check 方法，还是使用新版本方法创建一个 Permission 对象并调用 checkPermission()都会产生相同的结果。为了保持安全管理器对版本 1.0 和 1.1 的向后兼容性，新版本中的 Java API 继续使用了老的方法。Java API 继续调用了 28 个老式的 check 方法。但是，在具体安全管理器类中，老的方法大部分都用新的 checkPermission()方法实现了。因此通过调用具体安全管理器的老式方法，JavaAPI 实际上间接地调用了 checkPermission()方法。

Java API 可能多次直接调用 checkPermission()。对于版本 1.2 及其以后的版本中引入的新的潜在不安全操作的概念，不存在老式的 check 方法。所以在这种情况下，Java API 将创建一个新的 Permission 对象，这个对象不存在与之相关的 check 方法。然后 Java API 将把这个 Permission 对象直接传给安全管理器的 checkPermission()方法。

在具体的安全管理器类中，方法 checkPermission()同样负责决定，是否允许将某个操作的任务委派给另一个方法。此方法 checkPermission()只是简单地调用了类 java.security.AccessController 中的静态方法 checkPermission()，并将这个 Permission 对象传

递给它，因此在使用具体安全管理器时，类 AccessController 才是真正负责执行安全策略的实体。

Java 中的所有上述改变都是支持向后兼容的，假如为版本 1.1 创建了一个安全管理器，它也可以在 1.2 版本中正常运行，也可以在版本 1.2 中创建一个自定义的安全管理器，这样可以创建一个不同的安全基础架构，从而满足具体安全管理器实现所不能解决的特殊的安全需要。然而利用内置在具体安全管理器中的灵活性和可扩展性，大多数人的安全需要都应该能被满足。

3.4　代码签名和认证

Java 安全模型的重要功能是支持认证，这是从 java 1.1 的 java.security 包及其子包中就开始引入的特性。认证功能加强了用户的能力，使用户能通过实现一个沙箱来建立多种安全策略，这个沙箱可以依赖于为这个代码提供担保的对象来改变。认证可以使用户确认，由某些团体担保的一些 class 文件是值得信任的，并且这些 class 文件在到达用户虚拟机的途中没有被改变。这样，如果用户在一定程度上信任这个为代码作担保的团体，也就可以在一定程度上简化沙箱对这段代码实施的限制。可以对由不同团体签名的代码建立不同的安全限制。在本节的内容中，将简要介绍代码签名和认证的基本知识。

3.4.1　代码签名和密钥

要对一段代码作担保或者签名，必须首先生成一个“公钥/私钥”对。用户应该保管那把私钥，而把公钥公开。至少，应该把公钥给那些要你在你的签名上建立安全策略的人。一旦拥有了一个公钥/私钥对，就必须将要签名的 class 文件和其他文件放到一个 JAR 文件中，然后使用一个工具(例如版本 1.2 SDK 中的 Jarsigner)对整个 JAR 文件进行签名。这个签名工具将首先对 JAR 文件的内容进行单向散列计算，以产生一个散列。然后这个工具将用私钥对这个散列进行签名，并且将经过签名后的散列加到 JAR 文件的末尾。这个签名后的散列代表了你对这个 JAR 文件内容的数字签名。当你发布这个包含签名散列的 JAR 文件时，那些持有你的公钥的人将对 JAR 文件验证两件事：这个 JAR 文件确实是你签名的，并且在你签名后这个 JAR 文件没有做过任何改动。

数字签名过程的第一步是一个单向的散列计算，它输入大量的数据，但产生少量的数据，称为散列。在这个 JAR 文件的例子中，这个计算的大量输入就是组成这个 JAR 文件内容的字节流。这个单向散列计算之所以被称为“单向”，是因为在只给出散列的情况下，这个散列值不能包含足够的输入信息，因此不能从散列重新生成输入。这个计算时单向的，从大到小，从输入到散列。

散列也被称为消息文摘，它相当于一种束缚“指纹”。虽然不同的输入可能产生相同的散列，但通常认为，在实际的情况下，一个散列足以代表了它的输入。就像用指纹代表人一样，一个散列也被用于识别用单向散列算法产生这个散列的输入。在认证过程中，散列别用于验证某个输入是否和产生这个原始散列的输入相同，换句话说，这个输入在到达

目的地的途中有没有被改动。

因为不可能仅仅用散列重构原输入，一个散列尽在可以得到原输入时才有用。因此必须一起传输输入和散列。其实输入和散列的组合并不安全，因为就算是一个初级黑客也可以方便地将输入和散列一起替换掉。为了防止这种情况发生，必须在发送散列前使用私钥对其进行加密，只加密散列而不是对整个 JAR 进行加密，这是因为用私钥进行加密是一个相当费时的过程。一般来说，从 JAR 文件中计算、产生一个单项散列，并对这个散列用私钥进行加密，要比对整个 Jar 文件进行私钥加密来得快。只有当一个黑客拥有我们的私钥时，他才能同时替换输入和加密后的散列。由此可见，只要小心保护我们的私钥，破解输入和加密散列组合就更困难了，因为黑客不可能拥有我们的私钥。

任何用我们的私钥加密的东西都可以用我们的公钥解密。公钥/私钥对具有这种特点，在仅给出公钥的情况下，要产生私钥是非常困难的。如果黑客不能得到我们的私钥，对他来说最好的选择就是试图将原输入替换为另一个输入，这个输入必须和原输入产生相同的散列值。例如，一个黑客想要在你的 Jar 文件中将一个 class 文件替换成另一个执行恶意动作的 class 文件，被修改的 Jar 文件产生一个不同的散列的几率是非常高的。但是这个黑客可以往 jar 文件中添加随机的数据，知道产生和原来一样的散列值。如果黑客可以产生这样一个可供选择的输入——既可以帮助黑客达到邪恶的目的，又可以产生和原来的输入一样的散列——这个黑客就不需要你的私钥了。因为这个黑客的输入产生了和你的输入一样的散列值，而且你已经用你的私钥对这个散列进行了签名，所以这个黑客只要简单地将 Jar 文件中的你签名后的散列加到他的输入后就可以了。怎样才能防止黑客采用这种方法呢？然而对于黑客来说，这样的方法会花去大量的时间，因此几乎是不可行的。

因为单向散列算法是从大量数据(输入)中产生少量数据(消息摘要或者散列)，所以不同的输入可能产生相同的散列。单向散列算法倾向于充分随机地分布相同散列的输入，从而使产生相同散列值的概率主要依赖于散列的大小。例如，如果使用了一个长 8 位的散列值，散列算法最多产生 256 个不同的散列值。如果有一个 JAR 文件，它的散列值是 100，然后你开始将这个 8 位的散列算法在其他 Jar 文件上运用，毫无疑问，每进行大约 256 次计算就可能得到一个值为 100 的散列。如果散列的位数越多，产生相同散列值的情况就越不可能发送。在实际情况中，普遍采用的是 64 位或 128 位的散列，通常认为这个长度已经足够了，这时要想从一个不同的输入中产生一个相同的散列的计算是不可行的，因此防止黑客用恶意输入替换我们的善意输入，并且产生相同的散列值的主要障碍是他必须花费大量的时间和资源才能找到这个恶意的输入。

在产生散列值并用私钥对它签名之后，随后一个步骤就是将这个加密后的散列值加到同一个 Jar 文件中，这个 Jar 文件还包含了你最初产生这个散列的文件。这样一个经过签名的 Jar 文件，就包含了输入——你要担保的 class 文件和数据文件——以及用你的私钥加密过的散列值(由输入产生)。加密的散列代表了你对在同一个 Jar 文件中的类和数据文件的数字签名。

要想认证一个已签名的 Jar 文件，接收者必须用公钥对签名散列进行解密，得到的结果应该和从 Jar 文件计算而得到的散列值相等。为了验证一个 Jar 文件在签名后未被改动，接收者只要对 Jar 文件的内容实施单向散列算法，就像在签名过程中所做的那样。(记住，

并没有对 Jar 文件的内容进行加密，所以任何人都可以看见它。你只是将一个数字签名加到了那个 Jar 文件中)如果得到的散列值和加密的散列值匹配，那么接收者就可以推断，你确实为 Jar 文件进行了担保。而且这个 Jar 文件的内容在加上你的签名后没有被改动过。这个 Jar 文件中包含的代码就可以被放在一个不严格的沙箱中，这个沙箱信任你的签名。

尽管最初在 Java 1.1 版本中引入的认证技术利用了可信赖的数学原理，但是数学并不能解决所有问题，其实 Java 的认证技术引发了许多问题。例如认证技术没有说明应该信任谁，以及对他们的信任程度等问题。由这个认证技术引起了另一个和公钥的发布有关的问题。虽然最初这看起来很奇怪，但是在认证技术中将公开公钥，这种假设本身就产生了一些安全问题。公钥发布的困难是无论采取何种通信方式，消息(即公钥)可能潜在地被篡改或偷偷替换。当我们访问页面时可能被截取并涂改了。

为了解决公钥发布的困难，特地建立了许多证书机构为这些公钥做担保。

3.4.2 代码签名示例

为了说明 Java 虚拟机的签名机制，在接下来的内容中，将通过一个简单的示例来说明为代码签名的运作流程。在本示例中有三个类型：Doer、Friend 和 Stranger。其中第一个类型 Doer 定义了另外两种类型(类 Friend 和类 Stranger)实现的接口：

```
packagecom.artime.security.doer;
public interface Doer {
     void doYourThing();
}
```

Doer 仅声明了一个方法 doYourThing()，类 Friend 和类 Stranger 用相似的方式实现了这个方法。实际上除了名字不同以外，在本质上这两种方法是一样的：

```
packagecom.artime.security.friend;
importjava.security.AccessController;
importjava.security.PrivilegedAction;
importcom.artime.security.doer.Doer;
public class Friend implements Doer {
     private Doer next;
     private boolean direct;
     public Friend(Doernext, boolean direct) {
          this.next = next;
          this.direct = direct;
     }
     @Override
     public void doYourThing() {
          if (direct) {
               next.doYourThing();
          }else {

AccessController.doPrivileged(newPrivilegedAction<Friend>() {
                    public Friend run() {
                         next.doYourThing();
                         return null;
                    }
               });
```

```
            }
        }
}
packagecom.artime.security.stranger;
importjava.security.AccessController;
importjava.security.PrivilegedAction;
importcom.artime.security.doer.Doer;
public class Stranger implements Doer {
     private Doer next;
     private boolean direct;
     public Stranger(Doernext, boolean direct) {
          this.next = next;
          this.direct = direct;
     }
     @Override
     public void doYourThing() {
          if (direct) {
               next.doYourThing();
          }else {

AccessController.doPrivileged(newPrivilegedAction<Stranger>() {
                    public Stranger run(){
                         next.doYourThing();
                         return null;
                    }
               });
          }
     }
}
```

上述三种类型 Doer、Friend 和 Stranger 是为了说明访问控制的栈检查机制而专门推出的。在本章后面将给出一些栈检查的例子，到那时读者就会真正理解推出它们的目的。在这里通过编译 Friend 和 Stranger 而产生的 class 文件必须被签名，目的是以便在以后的栈检查的例子中使用它。从 Friend.java 产生的 class 文件将由一个比较信任的称为“friend”的团体签名，而从文件 Stranger.java 产生的 class 文件将由一个不太信任的称为“stranger”的团体签名，而由 Doer 产生的 class 文件不用签名。

在这些文件被签名以前，我们必须将它们放入 Jar 文件中。因为 Friend 和 Stranger 的 class 文件将被两个不同的团体签名，所以它们将被放置在两个不同的 Jar 文件中。通过编译文件 Friend.java 产生的两个 class 文件 Friend.class 和 Friend$1.class，将被放置在一个名为 friend.jar 中；同样由编译 Stranger.java 产生的两个 class 文件 Stranger.class 和 Stranger$1.class，将被放到一个名为 stranger.jar 的 Jar 文件中。

Friend.java 的 class 文件被 javac 编译器放到了目录 security/ex2/com/artima/security/friend 下，因为类 Friend 在包 com.artima.security.friend 中被声明，文件 Friend.java 的 class 文件必须被放置在 com/artima/security/friend 目录下的 Jar 文件中。在 security/ex2 目录下执行下面的命令，将会把 Friend.class 和 Friend$1.class 放到一个新建的名为 friend.jar 的 Jar 文件中，此 Jar 文件被放在了当前目录 security/ex2 下：

```
jar cvf friend.jarcom/artima/security/friend/*.class
```

当执行完上面的命令后，必须删除 Friend.java 的 class 文件，以便 Java 虚拟机在运行下面的访问控制的例子时无法找到它：

```
rmcom/artima/security/friend/Friend.class
rmcom/artima/security/friend/Friend$1.class
```

Stranger.java 的 class 文件被 javac 放在了 security/ex2/com/artima/secutiry/stranger 目录下，将它放入一个 Jar 文件的过程和上面的过程相同。在 security/ex2 目录下执行如下命令：

```
jar cvfstranger.jar com/artima/security/stranger/*.class
rmcom/artima/security/stranger/Stranger.class
rmcom/artima/security/stranger/Stranger$1.class
```

为了用 jarsigner 工具对 Jar 文件进行签名，在 keystore 文件中必须存储签名者个“公钥/私钥”对，这个文件用来存储已命名的、受密码保护的密钥。使用 keytool 程可以生成新的密钥对，并且将这个密钥对和一个名称相关联，并且用密码将它们保护起来，这个别名在每一个 keystore 文件中都是独立的，用于在一个特定的 keystore 文件中识别一个特定的密钥对。要想访问或者修改包含在 keystore 文件中的密钥对的信息，必须拥有这个密钥对的密码。

这个访问控制的例子需要在 security/ex2 目录下的名为 ivijmkeys 的 Keystore 文件，这个文件中包含别名为“friend”和“stranger”的两个密钥对。在 security/ex2 目录下运行下面的命令，将为别名 friend 产生密码为 friend4life 的密钥对。在这个过程中将产生一个名为 ijvmkeys 的 keystore 文件：

```
Keytool -genkey -alias friend -keypass friend4life-validity 10000 -
keystore ijvmkeys
```

在上述 keytool 命令中，命令行参数“-validity 10000”表示这个密钥对将在 10000 天之内有效。当命令运行时，它将产生一个 keystore 密码，在对这个 keystore 文件进行任意访问或修改时都需要这个 keystore 密码，赋给 ijvmkeys 的密码是 ijvm2ed。

我们可以用如下类似的命令为 stranger 生成密钥对：

```
Jarsigner -keystore ljvmkeys -storepass ijvm2ed -keypass
stranger4lifestranger.jar stranger
```

为了对两个 Jar 文件进行签名，必须做上面的工作。值得注意的是，在现实中必须确保不要让那些意图不轨的人得到你的私钥，并要和他们保持距离。这就意味着你不能丢失这个 keystore 文件，必须记住密码。而且，还必须让那些试图用你的签名来让我们的代码访问它们的系统的人得到你的公钥。

3.5 策略机制和保护域

策略机制和保护域是实现虚拟机安全性的因素之一，本节将详细讲解策略机制和保护域的基本知识，为读者学习本书后面的知识打下基础。

3.5.1 分析 Java 的策略机制

在本书前面的内容中曾经提到过，沙箱安全模型的最大的优点之一是可以是用户自定义的。通过从 Java 1.1 版本就已经引入的代码签名和认证技术，使正在运行的应用程序可以对代码区分不同的信任度。通过自定义沙箱，被信任的代码可以比不可靠的代码获得更多的访问系统资源的权限。这就防止了不可靠代码访问系统，但是却允许被信任的代码访问系统并进行工作。Java 安全体系结构的真正好处在于，它可以对代码授予不同层次的信任度来部分地访问系统。

Microsoft 提供了 ActiveX 控件认证技术，它和 Java 的认证技术相类似，但是 ActiveX 控件并不在沙箱中运行。这样使用了 ActiveX，一系列移动代码要么是被完全信任的，要么是完全不被信任的。如果一个 ActiveX 控件不被信任，则它将被拒绝执行。虽然这对于没有认证来说是一个很大的提高，但是如果一些恶意的或是有漏洞的代码得到了认证，这段危险的代码将拥有对系统的完全访问权。Java 的安全体系结构的优点之一就是，代码可以被授予只对它需要的资源进行访问的有限权限。即使一些恶意的或者有漏洞的代码得到了认证，它也很少有机会进行破坏。例如，一段恶意的或者有漏洞的代码可能只能删除一个固定目录下的为它设置的文件，而不是在本地硬盘上的所有文件。

从 1.2 版本开始的安全体系结构的主要目标是建立(以签名代码为基础的)细粒度的访问控制策略，这样不但过程更为简单而且更少出错。为了将不同的系统访问权限授予不同的代码单元，Java 的访问控制机制必须能确认应该给每个代码段授予什么样的权限。为了使这个过程变得容易，载入 1.2 版本或其他虚拟机的每一个代码段(每个 class 文件)将和一个代码来源关联。代码来源主要说明了代码从哪里来，如果它被某个人签名担保的话，是从谁那里来。在 1.2 版本以后的安全模型中，权限(系统访问权限)是授给代码来源的。因此如果代码段请求访问一个特定的系统资源，只有当这个访问权限是和那段代码的代码来源相关联时，Java 虚拟机才会把对那个资源的访问权限授予这段代码。

在 1.2 版本的安全体系结构中，对应于整个 Java 应用程序的一个访问控制策略是由抽象类 java.security.Policy 的一个子类的单个实例所表示的。在任何时候，每一个应用程序实际上都只有一个 Policy 对象。获得许可的代码可以用一个新的 Policy 对象替换当前的 Policy 对象，这是通过调用 Policy.setPolicy()并把一个新的 Policy 对象的引用传递给它来实现的。类装载器利用这个 Policy 对象来帮助它们决定，在把一段代码导入虚拟机时应该给它们什么样的权限。

安全策略是一个从描述运行代码的属性集合到这段代码所拥有的权限的映射。在 1.2 版本的安全体系结构中，描述运行代码的属性被总称为代码来源。一个代码来源是由一个 java.security.CodeSource 对象表示的，这个对象中包含了一个 java.net.URL，它表示代码库和代表了签名者的零个或多个证书对象的数组。证书对象是抽象类 java.security.Certificate 的子类的一个实例，一个 Certificate 对象抽象表示了从一个人到一个公钥的绑定，以及另一个为这个绑定作担保的人(以前提过的证书机构)。CodeSource 对象包含了一个 Certificate 对象的数组，因为同一段代码可以被多个团体签名(担保)。这个签名通常是从 Jar 文件中获得的。

从 1.2 版本开始，所有和具体安全管理器有关的工具和访问控制体系结构都只能对证书起作用，而不能对公钥起作用。如果附近没有证书机构，可以用私钥对公钥签名，生成一个自签名的证书。当使用 keytool 程序生成密钥时，总是会产生一个自签名的证书。例如在上一节的签名例子中，keytool 不仅产生了“公钥/私钥”对，而且还为别名 friend 和 stranger 产生了自签名的证书。

权限是用抽象类 java.security.Permission 的一个子类的实例表示的。一个 Permission 对象有三个属性，分别是类型、名字和可选的操作。权限的类型是由 Permisstion 类的名字指定的，例如 java.io.FilePermission、java.net.SocketPermission 和 java.awt.AWTPermission。权限的名字是封装在 Permission 对象内的。例如某个 FilePermission 的名字可能是 /my/finances.dat，某个 SocketPermission 的名字可能是 applets.artima.com:2000，某个 AWTPermission 的名字可能是 showWindowWithoutBannerWarning。Permission 对象的第三个属性是它的动作。并不是所有的权限都有动作。例如，FilePermission 的动作是“read,write”，SocketPermission 的动作是“accept,connect”。如果一个 FilePermission 的名字为/my/finances.dat，并且有动作“read,write”，那么它就表示对文件/my/finance.dat 可以进行读写操作，名字和动作都是由字符串来表示的。

Java API 有一个很大的权限层次结构，在里面表示了所有可能潜在危险的操作。可以根据自己的目的创建自己的 Permission 类来表示自定义的权限，例如可以创建一个 Permission 类来表示对属性数据库的特定记录的访问权限。定义自定义的 Permission 类也是一种扩展版本 1.2 的安全机制类满足自己需要的方法。如果创建了自己的 Permission 类，可以像使用 Java API 中的 Permission 类一样来使用它们。

在 Policy 对象中，每一个 CodeSource 是和一个或多个 Permission 对象相关联的。和一个 CodeSource 相关联的 Permission 对象被封装在 java.security.PermissionCollection 的一个子类实例中。类装载器可以调用 Policy.getPolicy()来获得一个当前有效的 Policy 对象的引用。然后它们可以调用 Policy 对象的 getPermission()方法，传入一个 CodeSource，从而得到和那个 CodeSource 对应的 Permission 对象的 PermissionCollection。类装载器然后可以使用这个从 Policy 对象中得到的 PermissionCollection 来帮助判断应该给导入的代码授予什么权限。

3.5.2　分析策略文件

Java.security.Policy 是一个抽象类，Policy 子类的实现细节之一就是该子类的实例怎样知道策略应该是什么。在此子类中可以采取多种方法，例如，对一个已序列化的 Policy 对象进行优化，从数据库中抽取策略，或者从文件中读取策略。在 Sun 提供的具体 Policy 子类中，在一个 ASCII 策略文件中用上下文无关文法描述安全策略。

一个策略文件包括了一些列 grant 子句，每一个 grant 子句将一些权限授给一个代码来源。我们知道一个代码来源包含了一个代码库和一系列签名，代码库是指出这个代码从那里下载的 URL。在策略文件中，签名用别名来代表，这些签名是保存 keystore 文件中的签名者的公钥。这个 keystore 可以在策略文件中用一个 keystore 语句显式说明。一个典型策略文件的例子是 security/ex2 目录下的文件 policyfile.txt：

```
Keystore "ijvmkeys";
grant signedBy "friend" {
            permission  java.io.FilePermission  "question.txt","read";
            permission  java.io.FilePermission  "answer.txt","read";
};
grant signedBy  "stranger"{
            permission  java.io.FilePermission  "question.txt", "read";
};
grant codebase "file:${com.artima.ijvm.cdrom.home}/security/ex2/*"{
            permission  java.io.FilePermission  "question.txt", "read";
            permission  java.io.FilePermission  "answer.txt", "read";
};
```

在上述 Policyfile.txt 文件中，其中第一条语句是 keystore 语句：

```
Keystore  "ijvmkeys";
```

上述 keystore 语句说明，密钥别名(策略文件其余部分将提到)指向存储在名为 ijvmkeys 的文件中的证书。因为这个文件名没有包含路径，这个文件必须是存储在当前目录下——java 应用程序使用该策略文件的启动目录。

上述策略文件中的第二条语句是一条 grant 语句：

```
grant signedBy "friend" {
            permission  java.io.FilePermission  "question.txt","read";
            permission  java.io.FilePermission  "answer.txt","read";
};
```

上述语句将授予由别名为 friend 的实体签名的所有代码两个权限。被授予的权限是：读取 question.txt 文件的权限，以及读取 answer.txt 文件的权限。因为这些文件名没有路径，所以它们必须是存储在当前目录下的，也就是这个应用程序的启动目录。因为这个 grant 子句中没有提到代码库，所以由 friend 签名的代码可以来自任何代码库。任何由 friend 签名的代码，不管是来自哪个代码库的，都将被授予对 question.txt 和 answer.txt 进行读操作的权限。

文件 Policyfile.txt 中的第三个语句也是一条 grant 语句，和上一条语句的形式类似：

```
grant signedBy  "stranger"{
            permission  java.io.FilePermission  "question.txt", "read";
};
```

上述语句将授予所有由别名为 stranger 的公司或个人签名的代码以下权限：读取名为 question.txt 的文件的权限。这个文件必须位于当前目录下，也就是这个应用程序的启动目录。因为在这条 grant 语句中没有提到代码库，所以来自所有代码库的代码，只要是由 stranger 签名的，都可以得到读 question.txt 的权限。注意，虽然 stranger 已经被允许读取 question.txt 中的问题，但是 stranger 不能看到 answer 中的答案。而授予 friend 的权限正好相反，它可以读取所有的问题以及答案。

在策略文件中的第四条，也是最后一条语句仍然是一个 grant 语句：

```
grant codebase "file:${com.artima.ijvm.cdrom.home}/security/ex2/*"{
            permission  java.io.FilePermission  "question.txt", "read";
            permission  java.io.FilePermission  "answer.txt", "read";
};
```

上述 grant 语句将两个权限授给所有从一个特定目录中装载的代码，对文件 question.txt 的读权限以及对文件 answer.txt 的读权限。这两个文件都必须在当前目录下，也就是这个应用程序的启动目录。这个 grant 子句没有指明任何签名者，所以这个代码可以是被任何人签名的，或者是未被签名的。只要它是从指定的目录下被装载的，它就可以被授予上面所列出的权限。

上述 grant 语句中的代码库 URL 采用了文件的形式，它包含了一个属性 ${com.artima.ijvm.cdrom.home}。

3.5.3　保护域

当类装载器将类型装入 Java 虚拟机时，会为每个类型指派一个保护域。保护域定义了授予一段特定代码的所有权限，一个保护域对应于策略文件中的一个或多个 Grant 子句。装载入 Java 虚拟机的每一个类型都属于一个、且仅属于一个保护域。

类装载器知道它装载的所有类或接口的代码库和签名者，利用这些信息来创建一个 CodeSource 对象。它将这个 CodeSource 对象传递给当前 Policy 对象的 getPermissions()方法，得到这个抽象类 java.security.PermissionCollection 的子类实例。此 PermissinCollection 包含了到所有 Permission 对象的引用，这些 Permission 对象由当前策略授予指定代码来源。利用它创建的 CodeSource 和从 Policy 对象得到的 PermissionCollection，可以实例化一个新的 ProtectDomain 对象。它通过将合适的 ProtectionDomain 对象传递给 defineClass()方法，来将这段代码放到一个保护域中。方法 DefineClass()是类 ClassLoader 的一个实例方法，用户自定义类装载器调用它来将类型导入到 Java 虚拟机中。将类型指派到保护域中是一个重要的工作，就像在本章前面提到的一样，它是类装载器体系结构支持 Java 沙箱安全模型的三个方法中的一个。

虽然这个 Policy 对象代表了一个从代码来源到权限的全局映射，但是最终还是由类装载器负责决定代码执行时将获得什么样的权限。例如一个类装载器可以完全忽略当前的策略，而随机地赋予权限。或者，一个类装载器可以向由 policy 对象的 getPermissions()方法返回的权限中再添加一些权限。例如，如果一个类型装载器要装载一个 applet 代码，除了由当前策略可能授予这段代码的权限以外，它还可以添加一个权限，使得它可以建立一个到这个 applet 的源主机的 socket 连接。现在读者可以明白了，类装载器在装载类时起到了重要的安全作用。

3.6　访问控制器

类 java.security.AccessController 提供了一个默认的安全策略执行机制，它使用栈检查来决定潜在不安全的操作是否被允许。这个访问控制器不能被实例化，它不是一个对象，而是集合在单个类中的多个静态方法。AccessController 的最核心方法是它的静态方法 checkPermission()，这个方法决定一个特定的操作能否被允许。这个方法将指向 Permission

对象的引用作为唯一的参数，并返回 void。和安全管理器中的 check 方法相类似，如果 AccessController 确定这个操作被允许，它的 checkPermission()方法将简单地返回；但是如果 AccessController 确定一个操作被禁止，它的 checkPermission()方法将异常中止，并抛出一个 AccessControlException，或者是它的一个子类。

在本章前面曾经提到过，安全管理器中的老式 check 方法(例如 checkRead()和 checkWrite())实现的只是简单的实例化一个合适的 Permission 对象，并且调用具体安全管理器的 checkPermission()方法。这个具体安全管理器方法 checkPermission()简单地调用了 AccessController 中的方法 checkPermission()。因此如果安装了具体安全管理器，其实最终是由这个 AccessController 来决定一个潜在不安全的方法是否被允许。

根据 AccessController 的 checkPermission()实现的算法，可以决定调用栈中的每个帧是否有权执行潜在不安全的操作。每一栈帧代表了由当前线程调用的某个方法，每一个方法是在某个类中定义的，每一个类又属于某个保护域，每个保护域包含一些权限。因此，每个栈帧间接地和一些权限相关。为了使传递给 AccessController 的 checkPermission()方法的 Permission 对象所代表的操作被允许，这个 AccessController 的基本算法要求，和调用栈上的每个帧相关联的权限必须包含或隐含传给 checkPermission()的 Permission 对象。

AccessController 的 checkPermission()方法自顶向下检查栈，只要它遇到一个没有权限的帧，它将抛出一个 AccessControlException。通过抛出这个异常，AccessController 指明这个操作不能被允许。相反，如果 checkPermission()方法到达栈的底部，也没有遇到这种栈帧(即无权限执行潜在不安全操作)的情况，checkPermission()方法将简单地返回。通过简单返回而不是抛出异常，AccessController 指明这个操作可以被允许。

3.6.1 implies()方法

为了决定由传递给 AccessController 的 checkPermission()方法的 Permission 对象所代表的操作，是否包含在(或隐含在)和调用栈中的代码相关联的权限中，AccessController 利用了一个名为 implies()的重要方法。这个 implies()方法是在 Permission 类以及 PermissionCollection 类和 ProtectionDomain 类中声明的。Implies()将一个 Permission 对象作为它唯一的参数，返回一个布尔值 true 或 false。Permission 类的 implies()方法确定由 Permission 对象所代表的权限，是否在本质上隐含在由一个不同的 Permission 对象所代表的权限中。PermissionCollection 和 ProtectionDomain 的 implies()方法确认了一个被传递的 Permission 是否包含或隐含在封装在 PermissionCollection 或 ProtectionDomain 中的 Permission 对象集合中。

例如，读取/tmp 目录下所有文件的权限本质上隐含了读取/tmp 目录下特定文件/tmp/f 的权限，反过来则不成立。如果你询问一个代表了读取/tmp 目录下的所有文件的权限的 FilePermission 对象，它是否隐含了读取文件/tmp/f 的权限，方法 implies()将返回 true。但是如果询问一个代表了读取/tmp/f 权限的 FilePermission 对象是否隐含了读取/tmp 下任何文件的权限时，方法 implies()将返回 false。

```
package com;
import java.io.File;
```

```
import java.io.FilePermission;
import java.security.Permission;
public class Example1 {
     public static void main(String[]args) {
          char sep = File.separatorChar;
          // readpermission for /tmp/f
          Permissionfile = new FilePermission(sep + "tmp" + sep + "f",
"read");
          // readpermission for /tmp/* which means all files in the /tmp
          // directory
          Permissionstar = new FilePermission(sep + "tmp" + sep + "*",
"read");
          boolean starImpliesFile= star.implies(file);
          boolean fileImpliesStar= file.implies(star);

          // print"Star   implies file = true"
          System.out.println("Starimplies file = " + starImpliesFile);
          // print"file   implies Star = false"
          System.out.println("file     implies Star = "
+fileImpliesStar);
     }
}
```

上述应用程序定义了类 Example1，在里面创建了两个 FilePermission 对象，一个代表对特定目录的读权限，另一个代表对同一个目录下的特定文件的读权限。这个从局部变量 star 引用的 FilePermission 对象代表了读取/tmp 下任何文件的权限，而从局部变量 file 引用的 FilePermission 对象代表了读取文件/tmp/f 的权限，在执行时，应用程序输出：

```
Star  impliesfile = true
file  impliesStar = false
```

方法 implies()被 AccessController 确定一个线程是否拥有进行某些操作的权限。例如，如果 AccessController 的 checkPermission()方法被调用，用以确定这个线程是否有权读取文件/tmp/f。AccessController 将调用和这个线程的调用栈中的每一个栈帧相关联的 ProtectionDomain 对象的 implies()方法。对于每一个 implies()方法，AccessController 将把一个 FilePermission 对象传递给它的 checkPermission()方法，这个 FilePermission 对象代表了读取文件/tmp/f 的权限。每个 ProtectDomain 对象的 implies()方法会调用它封装的 PermissionCollection 的 implies()方法，传递给它同一个 FilePermission。同样，每一个 PermissionColletion 会调用它包含的 Permission 对象上的 implies()方法，再一次传递这个 FilePermission 对象的引用。一旦 PermissinoCollection 的 implies()方法遇到了一个 Permission 对象，如果此 permission 对象返回 true，则这个 PermissionCollection 的 implies()方法也返回 true。只有当在 PermissionCollection 中包含的所有 Permission 对象的 implie()方法都没有返回 true 时，此 PermissionCollection 才返回 false。ProtectionDomain 的 implies()方法简单地返回了 PermissionCollection 的 implies()方法的返回值。如果 AccessController 从与一个特定栈帧相关联的 ProtectionDomain 的 implies()方法中得到 true 时，这个栈帧所代表的代码就拥有了执行这个潜在不安全操作的权限。

3.6.2 栈检查演示实例

接下来将给出几个演示实例，说明 AccessController 执行栈检查的方法。在下面的演示实例中，由 friend 和 Stranger 签名的代码在一定程度上被信任，但是使用 friend 签名的代码要比使用 stranger 签名的代码的可信度要高。由 friend 和 stranger 签名的代码都可以拥有读取文件 question.txt 的权限，在这个文件中包含了问题。由 friend 签名的代码可以拥有读取文件 answer.txt 的权限，在这个文件中包含了回答 question.txt 中的问题的答案，但是使用 stranger 签名的代码没有这个权限。下面的每一个例子将采取 policyfile.txt 中描述的策略。

```
package com.artime.security.doer;
public interface Doer {
    void doYourThing();
}
```

为了成为 Doer，类必须提供一个 doYourThing()方法的实现，实现 Doer 的类可以在它们的 doYourThing()方法中干任何它们喜欢的事。例如，这里有一个名为 TextFileDisplayer 的类，它实现了 Doer，它做的“事”是显示一个文本文件的内容：

```
packagecom.artime.security.doer.impl;
importjava.io.CharArrayWriter;
importjava.io.FileNotFoundException;
importjava.io.FileReader;
importjava.io.IOException;
importcom.artime.security.doer.Doer;
public classTextFileDisplayer implements Doer {
     private String fileName;
     publicTextFileDisplayer(String fileName) {
          this.fileName = fileName;
     }
     public void doYourThing() {
          try {
               FileReaderfr = new FileReader(this.fileName);
               try {
                    CharArrayWritercaw = new CharArrayWriter();
                    int c;
                    while ((c =fr.read()) != -1) {
                         caw.write(c);
                    }
                    System.out.println(caw.toString());
               }catch (IOException e){
                    e.printStackTrace();
               }finally{
                    try {
                         fr.close();
                    }catch (IOException e){
                         e.printStackTrace();
                    }
               }
          }catch(FileNotFoundException e) {
               e.printStackTrace();
```

```
            }
        }
}
```

在创建一个 TextFileDisplayer 对象时，必须将一个文件路径名传给它的构造器，这个 TextFileDisplayer 构造器将把这个路径名存储在名为 filename 的实例变量中。当调用这个 TextFileDisplayer 对象的 doYourThing()方法时，它将试图打开并读取这文件的内容，并把它们打印到标准输出上。

方法 doYourThing()的另一个例子来自类 Friend 和类 Stranger，它们在本章前面的代码签名的示例中已经出现过。

在 Friend 和 Stranger 中有很多相同的地方，它们有一样的实例变量、构造器以及 doYourThing()方法，它们仅仅是所在的包以及名称不同。当创建一个新的 Friend 或 Stranger 对象时，必须向构造器传递一个布尔值和到另一个对象的引用。这个构造器将传递进来的 Doer 引用存放在实例变量 next 中，并将布尔值存放在实例变量 direct 中。当一个 Friend 对象或一个 Strnager 对象的 doYourThing()方法被调用时，这个方法直接或间接地调用 next 中包含的 Doer 引用的 doYourThing()方法。如果 dirct 为 true，Friend 或 Stranger 仅仅直接调用 next 的 doYourThing()方法；否则 Friend 或 Stranger 的 doYourThing()通过一个 doPrivileged()调用，间接调用 next 的 doYourThing()方法。

第 4 章

通过网络实现移动性

长久以来，如何开发网络软件是 Java 开发人员所面临的最大挑战之一。在网络领域需要实现平台无关性，因为同一网络中通常连接了多种不同的计算机和设备。除此之外，安全模式也是一个挑战，因为网络可以方便地传输病毒和其他形式的恶意代码。本章将详细讲解 Java 如何把握网络所带来的巨大机遇，为学习本书后面的知识打下基础。

4.1 为什么需要网络移动性

在个人电脑流行之前，占主要地位的计算模式是服务于多个终端用户的大型计算机系统。大型主机利用分时技术分别关注从哑终端登录到主机的多个终端用户。软件应用程序存储在主机的磁盘上，使用户不仅可以共享一个 CPU，而且可以共享同样的应用程序。这种模式有个缺点，如果某个用户运行的作业占用了大量 CPU 资源的话，其他用户的性能就会大受影响。

微处理器的出现推动了个人计算机的蓬勃发展。硬件的这种变化引起了软件的相应变化，各用户不再共享存储在主机上的软件应用程序，而是在各自的 PC 机上拥有自己的软件拷贝。每个用户在自己专用的 CPU 上运行软件，因此，这种计算模式解决了多用户共享同一主机 CPU 时间多带来的问题。

最初，个人计算机像一些独立的孤岛一样分别进行计算。居统治地位的软件模式是在孤立的个人计算机上运行孤立的软件。但是很快，个人计算机开始互联成网。因为个人计算机只为自己的用户服务，它解决了大型计算机系统中 CPU 分时的难题。但除非这些个人计算机连接成网络，否则它们就不能像大型计算机系统那样使多个用户共享集中存储的数据了。

当个人计算机互联成网变得越来越普遍的时候，另一种软件模式日益重要起来，即“客户机/服务器”模式。“客户机/服务器”模式将任务分为两部分，分别运行在两种计算机上；客户端进程运行在终端用户的个人计算机上，而服务器端进程运行在同一网络的另一台计算机上。客户端和服务器端的进程通过网络来回发送数据进行传输。服务器端进程通常只是简单地接受网络中客户端发来的数据请求命令，从中央数据库中提取需要的数据，并将该数据发送给客户端。而客户端在接到数据后，进行处理，然后显示并允许用户操纵数据。这样的模式允许个人计算机的终端用户读取并操作放在中央储藏库的数据，而不需强迫这些用户共享中央 CPU 来处理数据。终端用户地区是共享了运行服务器端进程的 CPU，但在一定程度上，数据处理是由客户端完成的，因此服务器端 CPU 的负载大大减轻了。

很快，“客户机/服务器”模式中就不止包括两个处理器了。最初它被称做两层客户机/服务器模式：一层是客户端，另一层是服务器。更复杂一些的模型叫做三层(表示有三个进程)、四层(四个进程)或者 N 层结构，也就是说层次结构越来越多了。当更多的进程加入计算时，客户端和服务器的区别模糊了，于是人们开始使用“分布式处理”这个新名词来涵盖所有这些结构模式。

分布式处理模式综合了网络和处理器发展的优点，将进程分布在多个处理器上运行，并允许这些进程共享数据。尽管这种模式有许多大型计算机系统所无法比拟的优势，但它也有个不可忽视的缺点：分布式处理比大型计算机系统更难管理。在大型计算机系统中，软件应用程序存储在主机的磁盘上，虽然可以有多个用户使用该软件，但它只需在一个地方安装和维护。升级一个软件后，所有用户在下一次登录并启动该软件的时候可以得到这

个新的版本。但是相反，在分布式系统中，不同组件的软件往往存储在不同的磁盘上，因此，系统管理员需要在分布式系统的不同组件上安装和维护软件。要升级一个软件时，管理员不得不分别升级每台计算机上的这个软件。所以，分布式处理的系统管理比大型计算机系统要困难得多。

Java 的体系结构使软件的网络移动性成为可能，同时也预示了一种新的计算模式的到来。这种新的模式建立在流行的分布式处理模式的基础上，并可以将软件通过网络自动传送到各台计算机上。这样就解决了分布式处理系统中系统管理的困难。例如在一个 C/S 系统中，客户端软件可以存储在网络中的一台中央计算机上，当终端用户需要用该软件的时候，这个中央计算机会通过网络将可执行的软件传送到终端用户的计算机上运行。

因此，软件的网络移动性标志着计算模式发展历程中的重要一步，尤其是它解决了分布式处理系统中系统管理的问题，简化了将软件分布在多台 CPU 上的工作。它使数据可以跟相关软件一起传送。

4.2　网络对软件的影响

从大型计算机模式过渡到分布式处理模式是个人计算机革命的一个产物。而个人计算机革命的到来得益于处理器性能的快速增长和价格的降低。自从网络诞生那一天起，就形成了一种新的软件模式。本节将简要介绍这种模式的种种元素。

4.2.1　什么是网络

网络原指用一个巨大的虚拟画面，把所有东西连接起来，也可以作为动词使用。在计算机领域中，网络就是用物理链路将各个孤立的工作站或主机相连在一起，组成数据链路，从而达到资源共享和通信的目的。凡将地理位置不同，并具有独立功能的多个计算机系统通过通信设备和线路而连接起来，且以功能完善的网络软件(网络协议、信息交换方式及网络操作系统等)实现网络资源共享的系统，可称为计算机网络。

网络是信息传输、接收、共享的虚拟平台，通过它把各个点、面、体的信息联系到一起，从而实现这些资源的共享。网络是人们信息交流、使用的一个工具。作为工具，它一定会越来越好用，功能也会越来越多。内容也会越来越丰富。网络会借助文字阅读、图片查看、影音播放、下载传输、游戏聊天等软件工具从文字、图片、声音、视频等方面给人们带来极其丰富和美好的使用和享受。网络也是交流、资源共享的通道，但它毕竟是人类的一个工具，相信有一天，网络会借助软件工具的作用带给人们极其美好甚至超越人体本身所能带来的感受。比如借助软件工具让人以极其真实的外貌、感觉进入网络平台，从生老病死、游戏娱乐、结婚生子等。但这些只是丰富了人们的生活而不能取代人们的生活，它只能模仿人的感受而不能取代人的感受。网上可以直接实现虚拟产品的交易，如文字、影视、音乐的购买、发送、传输、接收。

4.2.2 计算机网络的发展历史

随着 1946 年世界上第一台电子计算机问世后的十多年时间内，由于价格很昂贵，电脑数量极少。早期所谓的计算机网络主要是为了解决这一矛盾而产生的，其形式是将一台计算机经过通信线路与若干台终端直接连接，我们也可以把这种方式看作最简单的局域网雏形。

世界上最早的网络是由美国国防部高级研究计划局(ARPA)建立的。现代计算机网络的许多概念和方法，如分组交换技术都来自 ARPAnet。ARPAnet 不仅进行了租用线互联的分组交换技术研究，而且做了无线、卫星网的分组交换技术研究，其结果导致了 TCP/IP 问世。

1977—1979 年，ARPAnet 推出了目前形式的 TCP/IP 体系结构和协议。1980 年前后，ARPAnet 上的所有计算机开始了 TCP/IP 协议的转换工作，并以 ARPAnet 为主干网建立了初期的 Internet。1983 年，ARPAnet 的全部计算机完成了向 TCP/IP 的转换，并在 UNIX(BSD4.1)上实现了 TCP/IP。ARPAnet 在技术上最大的贡献就是 TCP/IP 协议的开发和应用。2 个著名的科学教育网 CSNET 和 BITNET 先后建立。1984 年，美国国家科学基金会 NSF 规划建立了 13 个国家超级计算中心及国家教育科技网，随后替代了 ARPANET 的骨干地位。1988 年 Internet 开始对外开放。1991 年 6 月，在连通 Internet 的计算机中，商业用户首次超过了学术界用户，这是 Internet 发展史上的一个里程碑，从此 Internet 成长速度一发不可收拾。

4.2.3 网络应用形成了一种新的软件模式

软件模式向着具有网络移动性的分布式处理的方向发展，这得益于另一种硬件的发展，即网络带宽的性能提高和价格下降。网络带宽是指网络中可负载的信息总量。带宽的增长使传输新的数据成为可能；而每增加一种传输信息，网络就会呈现一种新的特性。这样，随着带宽的增长，网络上传输的简单文本可以附带上图片，网络就实现了报纸和杂志的功能。一旦带宽足以负载音频数据流，网络就能够承担收音机、CD 播放机或者电话的任务。带宽继续增长的话，传输视频数据就成为可能，网络就可以与电视机、录像机匹敌了。除此之外，网络带宽的增长还促进了另一种事务的发展，即计算机软件。因为网络是由互相连接的处理器组成的，理论上讲，只要有足够的带宽，一个处理器就可以通过网络将代码发送到另一个处理器上执行。一旦软件可以像数据一样被传输，整个网络就仿佛一台计算机一样。

一旦软件可以通过网络传输，那么不仅网络，就连软件本身，也会呈现出一种新的特征。具有网络移动性的代码很容易确保终端用户拥有必备的软件来浏览和操纵网络传输的数据，因为软件可以随数据一起传输。在旧的模式中，用户启动本地磁盘上的可执行软件来浏览网络传输过来的数据，所以软件应用程序通常是一个与数据完全独立的实体。而新的模式中，由于软件和数据都是由网络传输的，软件和数据的区别已经不那么明显了，软件和数据被统称为“内容”。

当软件的本质发生了变化时，终端用户与软件的关系也随之变化。在有网络移动性以

前，终端用户不得不考虑软件应用程序的版本问题。软件通常是通过磁带，软件或者光盘等介质来发布的。如果想使用某个应用程序，终端用户必须先得到这种安装介质，将他们插入计算机附加的驱动器，运行安装程序，将安装介质上的文件拷贝到计算机硬盘上。不仅如此，用户经常需要为同一个应用程序多次重复该安装过程，因为软件经常会有新旧版本的更替(新版本解决了旧版本中的问题，代替旧版本，但同时也会带来新的麻烦)。当新版本软件发布时，终端用户不得不决定是否要升级该软件。如果要升级，就要重复安装过程。因此用户不得不考虑软件的版本问题，并花大量精力来更新软件。

在新的软件模式下，终端用户可以较少考虑软件的版本问题，而享受一种自动更新的“内容服务”。尽管传统软件的安装和升级对用户来说是一项需要慎重考虑的工作，但软件的网络移动性可以使软件的安装和升级自动完成。网络发布的软件不需要让用户知道断断续续的版本号，用户也不必决定是否升级，不必亲自动手去升级。网络发布的软件能够自动保持版本一致。终端用户不用再购买版本众多的软件应用程序，而只需要订购一个通过网络发布的并带有相关数据的内容服务的软件，然后，这个软件和数据就可以自动更新了。

一旦抛弃了就有的，有多个版本的软件，而采用交互的内容自动更新的软件发布方式，终端用户就会丧失一些控制权力。在旧的软件模式下，如果软件的新版本出现了严重的 bug，终端用户只要不升级就可以避免问题了。但在新的软件模式下，因为用户可能没有权利控制升级过程，所以，有可能在发现新版本的问题前就已经被安装了新版软件。

对于某些软件，尤其是那些庞大的、功能齐全的软件，终端用户更希望能保留权利，让自己决定什么时候，在哪里升级。因此，在某些情况下，软件提供商会通过网络发布内容服务软件的不同版本。至少，提供商会发布一个服务的两个版本：一个是 beta 版本，一个是正式发布版本。希望使用更新版本的终端用户可以订购 beta 版本的服务，其他用户可以订购正式版本——因为尽管正式版本不像 beta 版本那样有最新的特性，但是更稳定，健壮性更好。

对于某些内容服务，尤其是简单的软件，大多数终端用户不想因过多地考虑版本问题而增加软件的使用难度。终端用户不得不了解不容版本间的差别，并决定什么时候、是否要费劲地去升级这个软件。而没有分散成多个版本的额内容服务相比之下就容易使用多了，因为它们可以自动升级。因为用户无需维护，而只要简单实用即可，所以这样的内容服务看上去就像一个盛软件的器皿。

许多自动更新的内容服务与普通的家居器皿有两个相似的重要特征：有一个主要的功能和一个简单的用户接口。比如说烤箱，烤箱的主要功能就是准备烘烤食物，而且有一个简单的用户接口——你想使用烤箱时，并不想去读复杂的说明书吧？你希望简单地把面包放进去，关上烤箱，看着里面亮起橘黄色的微光，片刻后，你的面包就烤好了。如果不小心烤得太焦或者不够，你就希望能有一个旋钮，以便下次烤面包时做相应的调整。这就是烤箱的功能和接口。与之相似，许多内容服务的功能也很单一，用户接口简单易用。比如想在网络上订购一部电影，你肯定不想费心去考虑是否有合适的电影订购软件版本，也不愿意去安装这种软件。你只想打开订购电影的内容服务，通过简单的用户接口来订购你的电影。然后，就可以坐下来欣赏网上传输过来的电影和享用烤面包了。

内容服务的一个绝好例子就是万维网网页。当你看一个 HTML 文件时，可以将它看作某种程序的源文件；但是如果你把浏览器看作应用程序的话，那 HTML 文件就可以看作数据了。因此，源代码与数据间的区别不很清晰。同样，人们浏览万维网时，希望网页能自动地不断更新。人们不希望看到网页的多个版本，不希望费很大力气手工地升级到网页的最新版本。

今后，现在的许多媒体都将在某种程度上融入网络，转型到内容服务上去。收音机、电视机、电话、应答机、传真机、录像带出租店、报纸、杂志、书籍，甚至计算机软件……这一切都会受到网络发展的影响。正如电视机不能完全取代收音机一样，网络传输的内容服务也不可能完全替代现有的媒体。但内容服务有可能取代现有媒体某些方面的功能，使其作出相应的调整，并创造出一些新的媒体形式。

在计算机领域，内容服务模式也不能完全取代旧的软件模式。它只能替代旧模式的某些方面(它们更适合新的软件模式)，增加一些新的形式，促使旧的软件模式进行调整。

4.3 Java 体系对网络的支持

Java 体系结构对网络移动性的支持是和它对平台无关性和安全性的支持密不可分的。虽然平台无关性和安全性对网络移动性而言并非是必需的，但它们对网络移动性的实际应用有很大帮助。在本节的内容中，将简要讲解 Java 体系对网络的支持的知识。

4.3.1 对网络安全的支持

21 世纪随着人们的生活和工作越来越网络化，网络安全已渐渐成为人们关心的一个重要问题。要保证信息在网络环境下是安全的，就必须对信息进行加密、签名、验证等一系列的操作。而 Java 语言能够对网络安全通信进行很好的支持。

随着计算机技术的飞速发展，信息网络已经成为社会发展的重要保证。信息网络涉及国家的政府、军事、文教等诸多领域。其中存储、传输和处理的信息有许多是政府宏观调控决策、商业经济信息、银行资金转账、股票证券、能源资源数据、科研数据等重要信息。还有很多是敏感信息，甚至是国家机密。所以难免会吸引来自世界各地的各种人为攻击(例如信息泄漏、信息窃取、数据篡改、数据删添、计算机病毒等)。

为了保证网络中的信息是安全的，必须采用一些加密、数字签名、身份认证等安全策略来有效地防范网络安全。Java 语言在网络安全方面提供了很强大的技术支持，从而能够很有效地保护信息在网络中的保密性、完整性和可用性。

1. Java 语言安全性的系统结构

对于 Java JDK 来说，无论代码在本地还是在远端运行，都要对应一个安全策略(Security Policy)。安全策略定义了不同签名者、不同来源的一套权限控制策略(Permissions)，在权限控制中说明了对资源(如文件、目录、网络端口等)的访问权限，如图 4-1 所示。

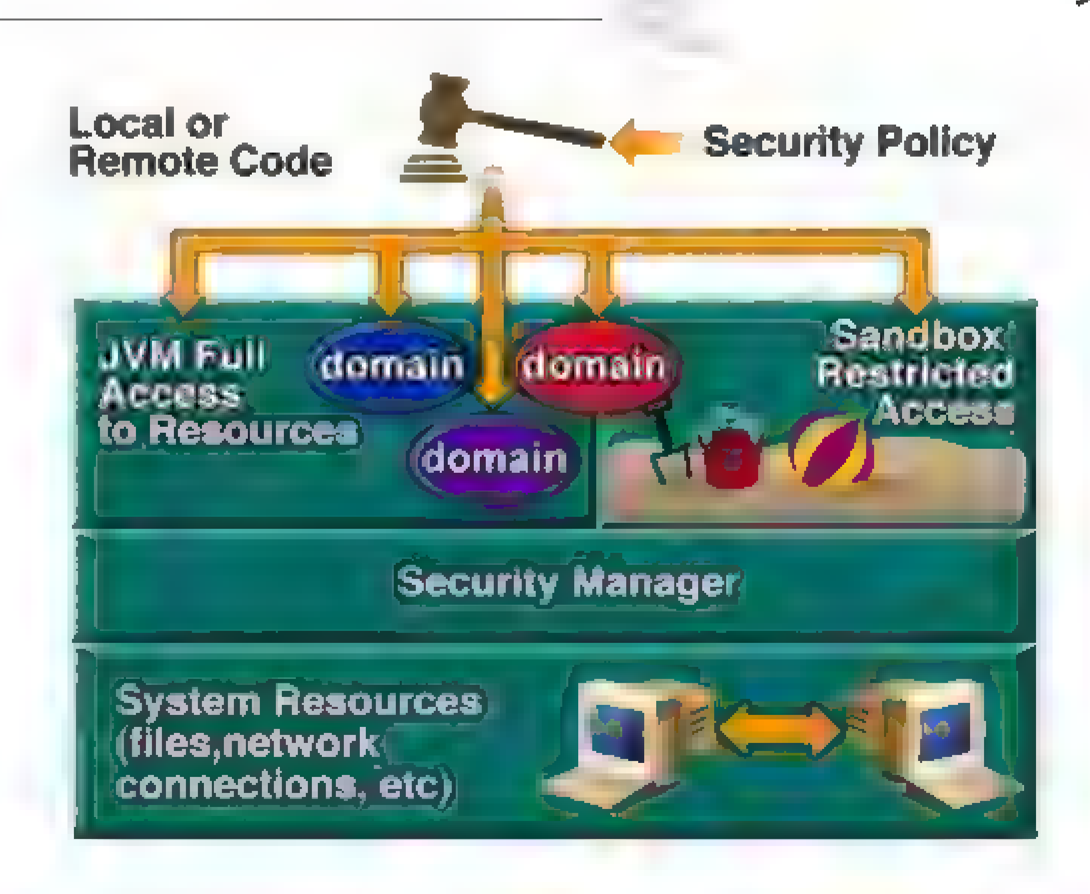


图 4-1　Java 的安全模型

运行系统将代码组织到单独的域(Domains)中，每个域封装一组具有相同控制权限的类的实例。域相当于沙箱(SandBox)，Java 小应用程序 Applet 只能在管理员的授权下运行于一个受限制的环境中。而应用程序则不用受这些限制，但必须在安全策略的授权下运行。

2. 密码使用的体系结构

从 JDK 6 版本开始，不但保留了以前的签名算法、消息摘要算法、密钥生成算法，还增加了密钥管理、算法参数生成、算法参数管理、随机数生成算法，支持不同密钥转化的代理和认证中心等安全性算法。JDK 中还增加了一个加密算法的扩展包(Java Cryptography Extension 即 JCE)，其模型如图 4-2 所示。它提供了全面的平台无关的安全应用 API 函数，实现了密钥管理、数字签名、MD5、SHA-1、基于 X.509 的认证代理等网络安全的常用功能。

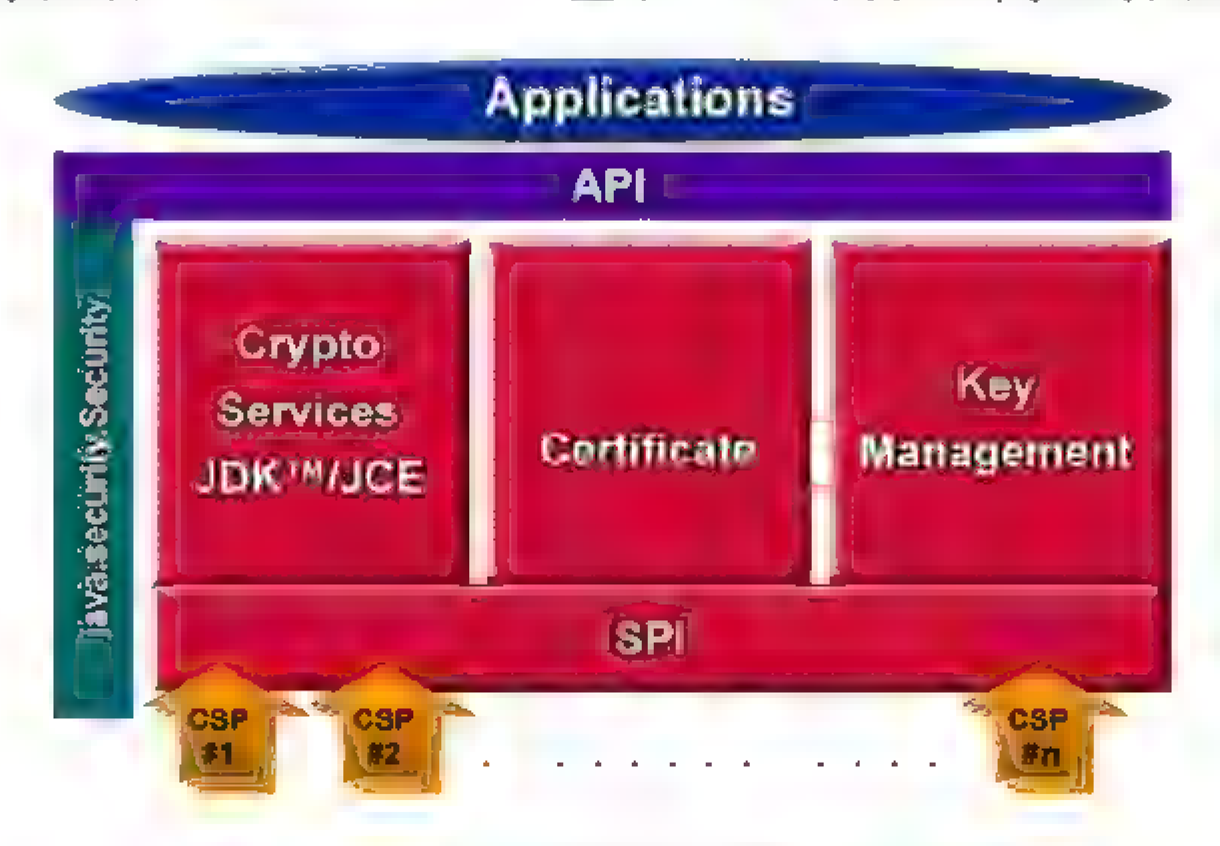


图 4-2　Java 加密算法扩展包(JCE)模型

3. 使用 Java 安全工具进行安全通信

1) Java 安全工具简介

使用 Java 安全工具可以用来设置安全策略并在远程站点上创建工作在安全策略范围内的应用程序。其常用的安全工具有如下几种。

(1) 密钥和证书管理工具(keytool)

keytool 工具主要是负责公私钥对的生成、向 CA 认证中心发送证书申请、接受 CA 的回

复并记录信任的公钥与实体的对应表，维护密钥库(keystore)。使用此命令的基本形式是：

```
keytool command options
```

keytool 工具主要有如下几种命令：

- certreq：产生一个证书签名请求(Generate a Certificate Signing Request, CSR)给CA，由 CA 来认证自己的证书。
- delete：删除对应的密钥库的记录。
- export：将公钥证书输出到某个文件。
- genkey：在密钥库中存入私钥与公钥对，后者保存在一个自己签名的证书中。
- import：将一个信任的证书导入，或者是接到了 CA 的回复，将该证书取代原来密钥库中自己签名的证书。
- keypasswd：为某私钥分配密码。
- list：列出密钥库中所有入口。
- storepasswd：给密钥库分配密码。

(2) Java 文档处理工具(jar)

如果要对代码签名，需要先用 jar 将其打包，然后用 jarsigner 来签名，使用命令的基本格式是：

```
jar cf jar-file input-file(s)
```

(3) Java 文档签名及验证工具(jarsigner)

jarsigner 工具通过密钥库中的数据来对 jar 文件进行签名和认证。

(4) 策略编辑器(policytool)

用于编辑系统的策略文件 policy。

2) 发送者签名并发送代码文件

发送方可对自己的 Java 代码进行签名后发送，以保证程序的安全性，其过程如图 4-3 所示。

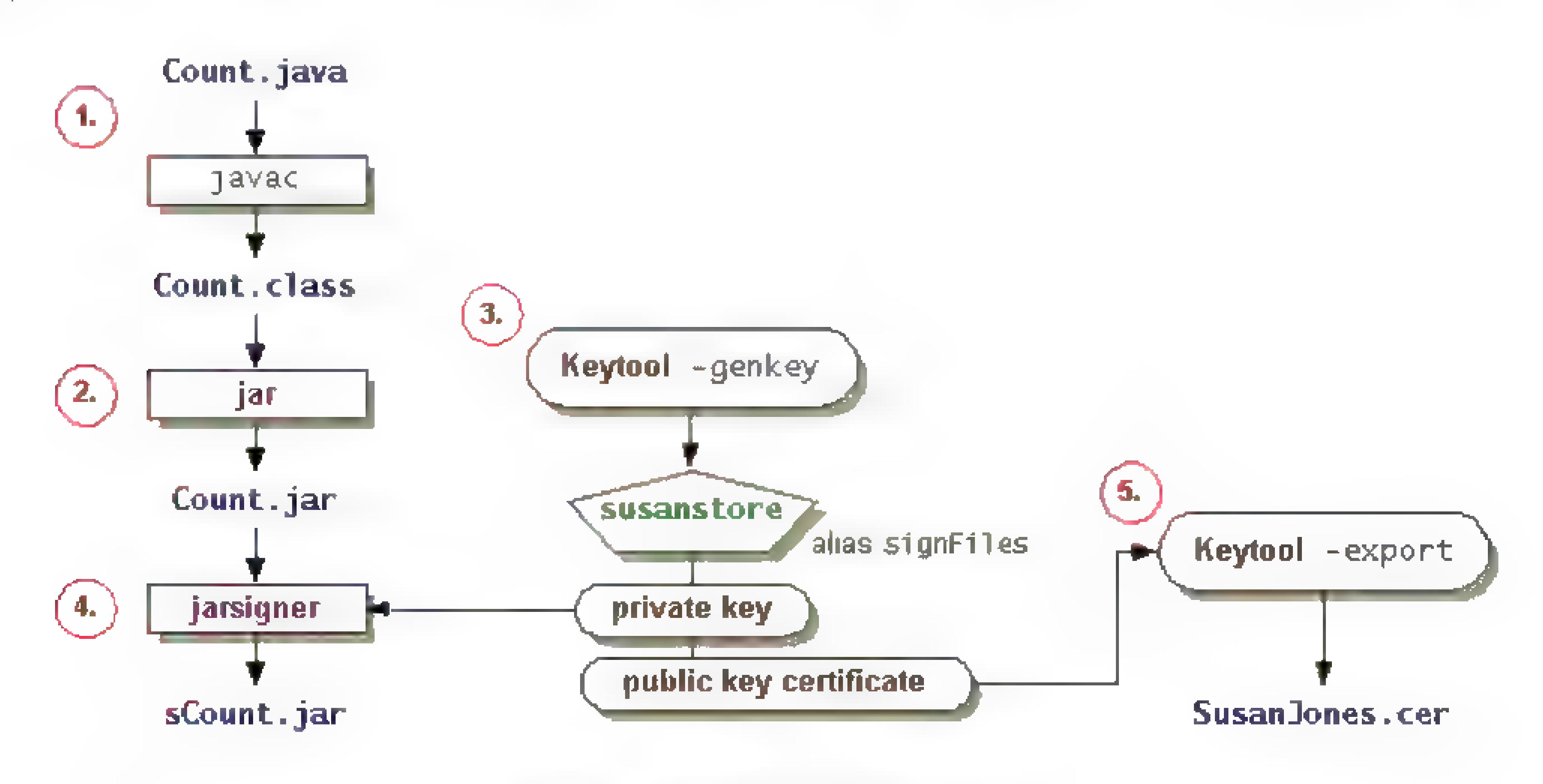


图 4-3　发送方签名代码文件

首先发送者可以将 Java 源程序编译为 class 文件，再将其压缩为 jar 文件，而后对压缩的 jar 代码文件进行数字签名。签名时使用 keytool 工具分别产生一对公私钥对，发送者使用私钥对 jar 代码文件进行签名，接受者就可以使用公开的公钥对签名过的 jar 文件进行验证。

发送者代码签名的具体过程如下：

(1) [生成私钥] keytool -genkey -alias signFiles -keypass 123456 -keystore store -storepass 123456

(2) [打包] jar cvf algrim.jar *.class

(3) [签名] jarsigner -keystore store -signedjar sAlgrim.jar Algrim.jar signFiles

(4) [成证书] keytool -export -keystore store -alias signFiles -file cer.cer

3) 接收者接受并验证代码文件

接收者在收到发送者签名的代码文件后可对其进行检验，以检查代码文件的机密性和完整性，其过程如图 4-4 所示。

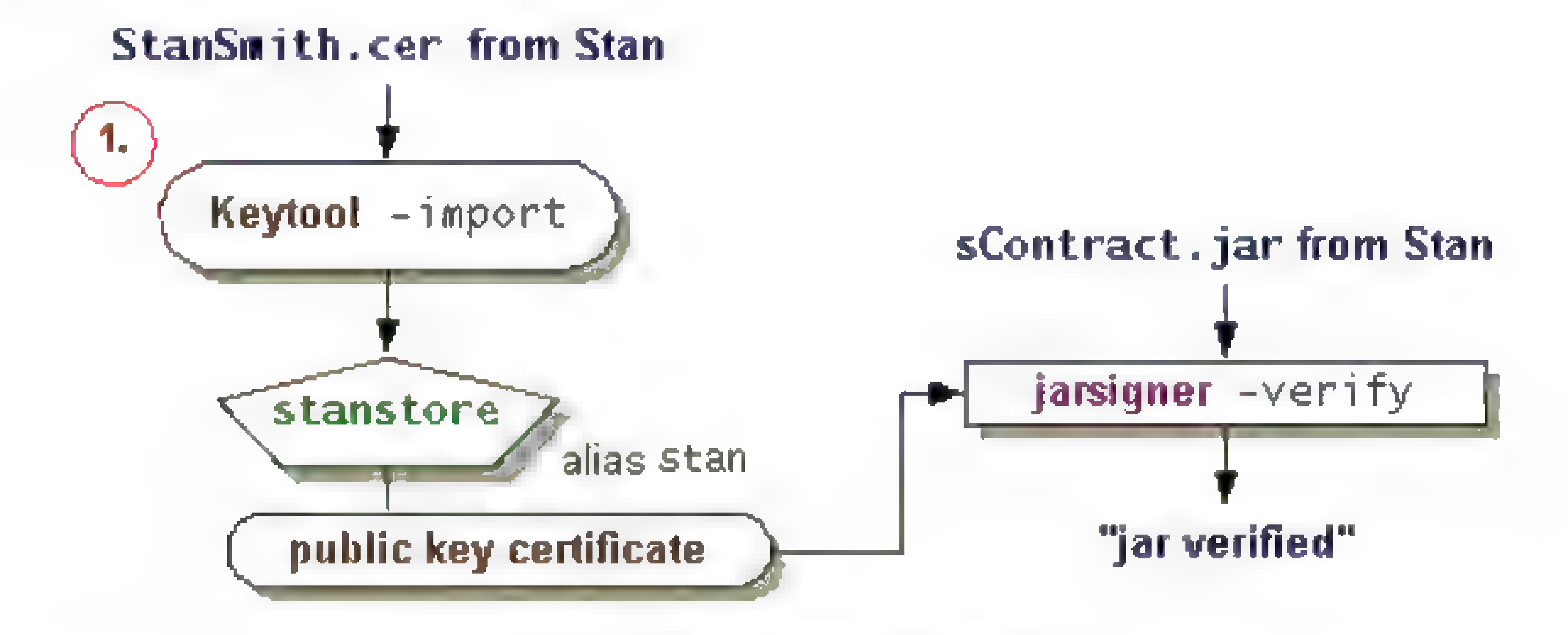


图 4-4　接收者检验代码文件

接收者在检验是使用签名者的公钥证书对签名后的代码文件进行验证。

4) 接收者安全运行代码文件

接收方在收到发送方签名的代码后直接运行其代码是不允许的，只有先将证书引入本地的 keystore 中，作为信任的证书插入，通过使用 Policy Tool 来配置相应的策略文件，才能够有效地运行发送的程序代码，其过程如图 4-5 所示。

运行时可使用以下两种配置方式。

(1) 使用如下形式：

```
java -Djava.security.manager -Djava.security.policy= raypolicy -cp
sCount.jar  AppName
```

(2) 通过配置浏览器使用的 java.home\lib\security\java.security 文件可以通过安全检查后运行该程序。

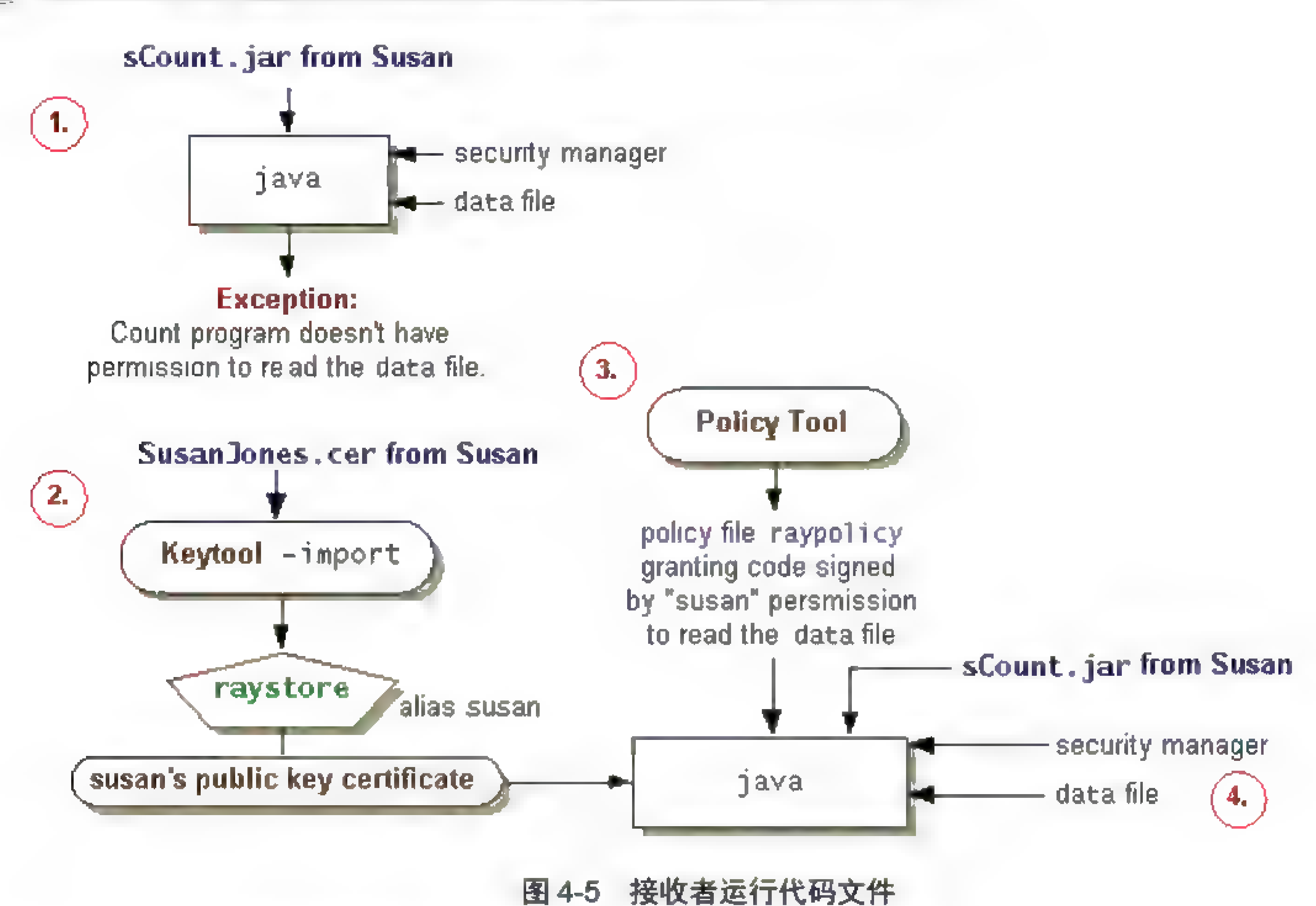


图 4-5 接收者运行代码文件

4. 构造自己的安全应用程序

1) 基础安全 API

在 Java JDK 中的类库中已提供了大量的类和接口来支持安全性程序的开发。通过这些类可方便构建自己的安全应用程序。

在 Java 包中的对安全性支持的接口(Interface)有：

- java.security.Certificate：指定类型的证书。
- java.security.Key：让书的密钥。
- java.security.Principal：可以提供 identify 的任何实体。
- java.security.PrivateKey：提供私有密钥。
- java.security.PublicKey：提供公有密钥。

在 java.security.acl 中还提供了如下支持访问控制的接口。

- java.security.acl.Acl：多个 acl 的入口的集合。
- java.security.acl.AclEntry：acl 的入口。
- java.security.acl.Group：一组 Principal。
- java.security.acl.Owner：acl 的管理者。
- java.security.acl.Permission：控制信息。

2) 构造安全应用程序的一般过程

使用 Java JDK 来构造安全应用程序的一般过程如下。

(1) 产生公钥和私钥对

[得到密钥产生器] KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA",

"SUN");

[初始化密钥产生器] SecureRandom random = SecureRandom.getInstance("SHA1PRNG", "SUN"); keyGen.initialize(1024, random);

[产生公钥和密钥] KeyPair pair = keyGen.generateKeyPair();PrivateKey priv = pair.getPrivate(); PublicKey pub = pair.getPublic();

(2) 对数据进行签名

[得到一个签名对象] Signature dsa = Signature.getInstance("SHA1withDSA", "SUN");

[初始化签名对象] dsa.initSign(priv);

[对数据签名] dsa.update(buffer, 0, len);

[得到签名的数据] byte[] realSig = dsa.sign();

(3) 存储签名和公钥

签名结果直接按字节流存储；公钥通过 pub.getEncoded()，先转换为字节流来处理。

(4) 从文件中取得公钥

[从文件中读到字节流中] encKey

[构造一个密钥说明类] X509EncodedKeySpec pubKeySpec = new X509EncodedKeySpec(encKey);

[构造一个密钥管理器] KeyFactory keyFactory = KeyFactory.getInstance("DSA", "SUN");

[取得公钥] PublicKey pubKey = keyFactory.generatePublic(pubKeySpec);

(5) 验证签名

[同生成签名一样先取得签名对象] Signature sig = Signature.getInstance("SHA1withDSA", "SUN");

[用公钥初始化签名对象] sig.initVerify(pubKey);

[取得被签名的数据] sig.update(buffer, 0, len);

[验证] boolean verifies = sig.verify(sigToVerify);

4.3.2 网络移动性

平台无关性使得在网络上传送程序更加容易，因为不需要为每个不同的主机平台都准备一个单独的版本，因此也不需要判断每台计算机需要哪个特定的版本，一个版本就可以对付所有的计算机。Java 的安全特性促进了网络移动性的推广，因为最终用户就算从不信任的来源下载 class 文件，也可以充满自信。因此实际上，Java 体系结构通过对平台无关性和安全性的支持，更好地推广了它的 class 文件的网络机动性。

除了平台无关性和安全性之外，Java 体系结构对网络移动性的支持主要集中在对在网络上传送程序的时间进行管理上。假若你在服务器上保存了一个程序，在需要的时候通过网络来下载它，这个过程一般都会比从本地执行该程序要慢。因此对于在网络上传送程序来说，网络移动性的一个主要难题就是时间。Java 体系结构通过把传统的单一二进制可执行文件切割成小的二进制碎片——Java class 文件——来解决这个问题。class 文件可以独立在网络上传播，因为 Java 程序是动态链接、动态扩展的，最终用户不需要等待所有的程序

class 文件都下载完毕，就可以开始运行程序了。第一个 class 文件到手，程序就开始执行。class 文件本身也被设计得很紧凑，所以它们可以在网络上飞快地传送。因此 Java 体系结构为网络移动性带来的直接主要好处就是把一个单一的大二进制文件分割成小的 class 文件，这些 class 文件可以按需装载。

Java 应用程序从某个类的 main()方法开始执行，其他的类在程序需要的时候才动态链接。如果某个类在一次操作中没有被用到，这个类就不会被装载。比如说，假若你在使用一个字处理程序，它有一个拼写检查器，但是在你使用的这次操作中没有使用拼写检查器，那么它就不会被装载。

除了动态链接之外，Java 体系结构也允许动态扩展。动态扩展是装载 class 文件的另一种方式，可以延迟到 Java 应用程序运行时才装载。使用用户自定义的类装载器，或者 class 类的 forName()方法，Java 程序可以在运行时装载额外的程序，这些程序就会变成运行程序的一部分。因此，动态链接和动态扩展给了 Java 程序员一些设计上的灵活性，既可以决定何时装载程序的 class 文件——而这又决定了最终用户需要等待多长时间来从网络上装载 class 文件。

除了动态链接和动态扩展，Java 体系结构对网络移动性的直接支持还通过 class 文件格式体现。为了减少在网络上传送程序的时间，class 文件被设计得很紧凑。它们包含的字节码流设计得特别紧凑——之所以被称为“字节码”，是因为每条指令都只占据一个字节。除了两个例外情况，所有的操作码和它们的操作数都是按照字节对齐的，这使得字节码流更小。这两个例外是这样一些操作码，在操作码和他们的操作数之间会填上 1～3 个字节，并且在操作数时都按照字边界对齐。

class 文件的紧凑型隐含着另外一个含义，那就是 Java 编译器不会做太多的局部优化。因为二进制兼容性规则的存在，Java 编译器不能做一些全局优化，比如把一个方法调用转化为整个方法的内嵌(内嵌指把被调用方法的整个方法体都替换到发起调用的方法中去，这样在代码运行的时候，可以节省方法调用和返回的时间)。二进制兼容性要求：假若一个方法被现有的 class 文件依赖，那么改变这个方法的时候必须不破坏已有的调用方法。在同一个类中使用的方法可能使用内嵌，但是一般来说 Java 编译器不会做这种优化，部分原因是因为，这样为 class 文件瘦身得不偿失。优化常常是在代码大小和执行速度间进行的折中。因此，Java 编译器通常会把优化工作留给 Java 虚拟机，后者在装载类之后，再解释执行，即时编译或者自适应编译的时候都可以优化代码。

除了动态链接、动态扩展和紧凑的 class 文件之外，还有一些并非体系结构必须的策略，可以帮助控制在网络上传送 class 文件的时间。因为 HTTP 需要单独为 Java Applet 中用到的每一个 class 文件单独请求连接，那么我们会发现下载 applet 的很大一部分时间并不是用来实际传输 class 文件的时间，而是每一个 class 文件请求网络协议握手的时间。一个文件需要的总时间是按照需要下载的 class 文件的数目倍增的。为了解决这个问题，Java 1.1 包含了对 Jar 的支持，Jar 文件允许在一次网络传输过程中传输多个文件，这和一次次传送一个个单独的 class 文件相比，大幅度降低了需要的总体下载时间。更大的优点是：Jar 文件中的数据可以压缩，从而使下载时间更少。正是因为这个原因，所以有时候通过一个大文件来传送软件。例如有些 class 文件是程序开始运行之前所必需的，这些文件可以

很快地通过 Jar 文件一次性传递。

另外一个降低最终用户等待时间的策略就是不采取按需下载 class 文件的做法。有几种不同的技术，例如 MarimbaCastanet 使用的订阅模式，可以在需要 class 文件之前就把它们下载下来，这样程序就可以更快地启动。

因此，除了平台无关性和安全性能够对网络移动性有利外，Java 体系结构的主要着眼点就是控制 class 文件在网络上传送的时间。动态链接和动态扩展允许 Java 程序按照小功能单元设计，在最终用户需要的时候才单独下载。class 文件的紧凑性本身有助于减少 Java 程序在网络上传送的时间。Jar 文件允许在一次网络连接中传送多个文件，还允许数据压缩。

4.4　applet 演示

Java 是一种网络化的技术，在网络被逐渐证明成为下一次计算革命的时代出现。然而 Java 被如此迅速而广泛使用的原因，并不在于它是一种适时的技术，而是因为它有合适的市场。Java 并非是 20 世纪 90 年代中期开始发展的唯一一种基于网络的语言。虽然他是一个好技术，它并非一定是最好的——但是它可能是最有市场的。Java 在 1995 年初打开了一小块市场，结果获得了很强烈的回应，使得很多开发类似技术的公司被迫取消了他们的项目。拥有类似技术的公司，比如 AT&T，他们拥有一个网络化的技术 Inferno，不得不面对 Java 窃取了它们本可能获得的掌声这样一个事实。

Java 从最初产生到获得巨大的市场成功，有几个很重要的事实。首先，他有一个很酷的名字——除了程序员之外，非程序员也能赏识它。其次，它在所有的应用中都是免费的——对潜在的客户来说这是一个魔咒。但是 Java 获得市场成功最重要的一点就是，Sun 的工程师适时把 Java 技术和 WWW 融合了起来，那恰恰是 Netscape 试图把他们的网络浏览器从一个图形化的超文本查看器变成一个全功能计算平台的时候。随着 WWW 如同不断增高的巨潮席卷整个软件产业(也是一次全球思潮)，Java 就站在了浪尖上。因此，Java 的成功可以说是因为 Java 会“在网上冲浪”，它在正确的时间抓住了浪潮，并且一次次稳稳地站在浪尖上，它潜在的竞争者都被无声地吞没了。Sun 的工程师把 Java 技术和 WWW 融合起来的方法，这就是 Java 市场成功的关键所在，也就是创造了一种 Java 程序的特殊形式，可以在 Web 浏览器内部运行 Java Applet(Java 小应用程序)。

Java Applet 展示了 Java 基于网络的所有特性：平台无关性、网络移动性和安全性。平台无关性对于 WWW 来说是一个主要原则，Java Applet 正好符合。在任何平台上，只要有支持 Java 的浏览器，Java Applet 就可以运行。Java Applet 也展示了 Java 在安全上的能力，因为它们是在一个严格受限的沙箱中运行的。但是最重要的，Java Applet 展示了它承诺的网络移动性。如图 4-1 所示，Java Applet 可以在一个中心服务器上维护，可以通过网络传送到很多不同种类的计算机中。要升级一个 applet，只需要升级服务器上的即可。用户下一次使用 applet 的时候，就可以得到升级过后的版本。因此，维护是本地的，运行却是分布式的。



支持 Java 的浏览器取代了使用 Java 程序来包容浏览器显示类 applet 的方法。要显示一个网页，Web 浏览器从 HTTP 服务器请求一个 HTML 文件。加入的 HTML 文件中包含一个 applet，浏览器会看到下面的 HTML 标记：

```
<applet CODE="HeapOfFish.class"
CODEBASE="gcsupport/classes"
WIDTH="525"
HEIGHT=360>
</applet>
```

上面的“applet”标记会给浏览器足够的信息来显示这个 applet。CODE 属性标示了 applet 初始 class 文件的名字，这里是 HeapOfFish.class。CODEBASE 属性指明了 class 文件相对于这个网页的 URL 路径，WIDTH 和 HEIGHT 属性表明了 applet 屏幕的像素宽度和高度，也就是网页上 applet 会显示出来的区域。

当浏览器遇到一个包含 applet 标记的网页时，它把信息从标记送到一个运行中的 Java 程序。Java 程序创建一个新的用户自定义的类装载器对象(或者重用已有的那个)，来装载这个 applet 的出事 class 文件。然后它首先调用 applet 的初始类的 init()方法，再调用 start()方法，这样就初始化了 applet。applet 需要的其他 class 文件按照按需下载的原则在动态连接的处理过程中下载。比如说，当一个 applet 的初始类第一次用到某个新类的时候，必须解析新类的符号引用，在解析过程中，如果类早已被装载，Java 虚拟机会要求装载同一个用户自定义的类装载器(该装载器装载该 applet 的初始类以装载新类)。如果用户自定义的类装载器不能本地装载类，它就会尝试从网络上下载这个 class 文件。装载器将试图从它获得 applet 的初始类相同的位置下载。一旦 applet 的初始化完成了，applet 就如同网页的一部分一样在浏览器中显示出来。

4.5 JINI 服务对象

JINI(是 Java Intelligent Network Infrastructure 的缩写)是 Sun 公司的研究与开发项目，它能极大地扩展 Java 技术的能力。JINI 技术可使范围广泛的多种硬件和软件，即可以与网络相连的任何实体，能够自主联网。本节将简要讲解 JINI 的基本知识。

4.5.1 Java 推出 JINI 的背景

虽然 Java 体系结构使代码的网络移动性成为可能，并由代表了计算模型的一次重要革命的 Javaapplet 表现出来，但 Java 结构还有另外一个承诺：对象的网络移动性。对象在网络中穿梭，携带者定义自己的类，加上表示对象状态的快照数据。如同代码的网络移动性可以简化系统管理员的工作一样，对象的网络移动性可以简化软件开发者设计和部署分布式系统的工作。通过对象序列化和远程方法调用(RMI)，Java API 提供了一个在本地对象模型上扩展而成的分布式对象模型，打破了 Java 虚拟机之间的界限。分布式对象模型使得一个虚拟机中的对象可以引用另一个虚拟机中的对象，调用那些远程对象的方法，在虚拟机之间把对象当作参数、返回值或者方法调用抛出的意外来交换。这种由 Java 底层基于网

络的体系结构所带来的能力，可以简化设计分布式系统的任务，因为它们有效地把面向对象编程带入了网络。

通过 Java 网络友好的底层体系结构，以及对象序列化和 RMI 技术，有一种技术利用了对象网络移动性的全部好处，那就是 Sun 的 JINI 技术。JINI 是一系列协议和 API 的集合，可以支持对分布式系统的编写和部署，目标是把正在快速发展的、没有硬盘的嵌入设备连接到网络中。JINI 体系机构的一个特别组成部分是服务对象，它可以告诉我们对象移动性是多么有用。

JINI 系统是以“查找服务”为中心的，其他的服务在“查找服务”中注册，这是通过在对象之间传递一种特殊的“服务对象”来完成的。服务对象对客户机来说就代表服务。需要访问服务的客户机从查找服务中获取一个服务对象的拷贝，通过调用服务对象的方法来和服务进行交互。服务对象负责实现服务——不管是本地实现的服务，还是和网络另一端的软件进程或者硬件对话实现的服务。

提出 JINI 这一体系结构的目标是将成组的硬件设备和软件组件联合成一个单一的、动态的分布式系统，联合后的网络系统更加易于管理和使用，同时在保持单机的灵活性、统一响应和控制的情况下，还能够支持由系统提供的共享能力。JINI 这一体系结构中有几个非常重要概念。

1) 服务(Service)

服务是一个独立的功能实体，它可以被人、程序或者其他服务使用。服务这一概念在 JINI 中包括的内涵非常丰富，它可以是一次计算过程或者存储操作，也可以是和另一个用户交流的通道，甚至可以是一个硬件设备或者另一个用户。JINI 系统中成员间的联盟是为了对服务进行共享访问，一个 JINI 联盟不应被简单地看成是客户机和服务器的集合，而应当看作是组合到一起完成某个特定任务的服务集合。JINI 提供了相应的机制，能够在分布式系统中实现对服务的构造、查找、通信和调用，同时还提供了一套服务协议来负责服务间的通信。

2) 客户(Client)

JINI 中的客户是需要利用服务的硬件设备或软件组件，JINI 的目标是支持尽可能多的异构客户，包括各种硬件设备和软件平台。

3) 查找服务(Lookup Service)

是 JINI 中的一种服务协议，它允许软硬件发现网络并变成联盟中的成员，同时将所提供的服务广播给联盟中的其他成员。

4.5.2　什么是 JINI

在 JINI 的“思维”方式中，网络是由“服务”组成的，客户机或者其他服务可以利用这些服务。服务可能是网络上的任何形式，它们准备好实现某种功能。硬件设备、软件服务器、或者是通信信道，甚至是用户自己都可以成为服务。比如说支持 JINI 的磁盘驱动器，可以提供“存储”服务。支持 JINI 的打印机可以提供“打印”服务。

为了完成一项任务，客户机征用一些服务来帮助它。比如，客户程序可能从数码相机的图像存储服务中抓取(上传)图片，把它们下传到磁盘驱动器提供的永久存储服务中，把

一整页缩小的图片发送到彩色打印机提供的打印服务。在这个例子中，客户程序建立了一个分布式系统，包括程序本身、图像存储服务、永久存储服务，还有彩色打印服务。客户机和这个分布式系统中的各个服务协同工作来完成任务：把数码相机中的照片取出来，打印出一张缩略图。

4.5.3 为什么需要 JINI

JINI 可以使人们极其简单地使用网络设备和网络服务，就像今天我们使用电话一样，通过网络拨号即插即用。JINI 的目标是最大限度地简化与网络的交互性。

JINI 利用了 Java 技术的优势。JINI 包含了少量类库格式的 Java 代码和某些惯例，可在网络上创建一个 Java 虚拟机的“王国”，就像我们人类创造一个社区一样。在这个王国里的人、设备、数据和应用程序等网络公民均被动态地连接起来，从而能够共享信息和执行任务。

1) 趋势所趋

网络的普及这个世界正在网络化。例如，在今天，一个企业要想取得成功就必须建立网络。商业网络正在不断扩大，而且已经能够与供应商和客户实现直接交互。与无线网络的交互也几乎成为家常便饭。企业和消费者都要求能与网络进行更广泛的交流。出差在外的人无不希望在到达饭店后就能把自己的计算机插入网络接口，不但能与自己单位的工作环境进行交互工作，而且还能与饭店的本地服务，如打印机或传真机等进行交互工作。父母可能希望只需使用移动电话或笔记本电脑就能与家里的摄像机相连，通过它来察看家里的情况。人们无不希望随时随地能够连接和立即使用本地的定制服务。在不远的将来，我们将看到网络渗透到很多其他环境。例如，将会出现把电视机和立体声设备等音频/视频设备与家庭办公室的电脑和外设连接起来的网络，并控制安全监视器和温控恒温器等网络设备。电缆和 ASDL 等高带宽媒介将为家庭提供全新的服务。服务供应商不断为驾驶员提供越来越多的服务，网络也必将随之进入汽车领域。除导航系统外，游览景点和当地餐馆名单等本地服务也将出现在驾驶员的屏幕上。只要汽车与远程诊断设备相连，它就能自动完成对汽车的维护，并在汽车出现问题时通知驾驶员。商业机遇——网络服务 JINI 所能带来的商业机遇是新型的网络服务。

例如，产品制造商将在基于网络的产品上提供新的服务。例如，磁盘可被看作与网络相连的存储服务，能向磁带和其他新型服务提供自动存储备份。联网的摄像机可能将提供诸如安全监视等新型成像服务。这些新的服务使制造商成为新型的网络服务供应商。

2) JINI 能帮助传统的服务供应商提供新型服务

例如，某媒体服务供应商可能希望向某消费者的家庭打印机提供报纸打印服务。无线服务供应商可能希望通过蜂窝电话提供相似的服务。

3) JINI 可以简化对现有服务的管理

在隔天交货的情况里，JINI 简化了分布在各处的工人与网络连通的方式。在个人银行里，基于 JINI 的计算机和外设可简化分行的系统管理。对于无线服务供应商，JINI 可使蜂窝电话具备类似于电话的网络功能：屏幕大小、处理能力、使所提供的服务根据每一部电话的特点而专门设计。问题是，在今天的环境中，联网还是太复杂了。例如，无论是把

PC 连接到网络上，还是使用联网的打印机都非常复杂。只有经验丰富的系统管理员才有能力处理装载驱动程序、设置配置文件等复杂的工作。显然，我们不可能指望一般消费者也能管理今天这样复杂的网络。

今天的网络还很脆弱和很不灵活。对网络稍加改动就可能造成不可挽救的大混乱。向网络中添加诸如磁盘存储等功能的过程也很复杂。例如，要想添加一个磁盘驱动器，我们就必须打开机箱，处理设置跳线器，并解决一系列复杂的设置问题。即使专家也会头疼。

从消费者的角度看，他们所需要的只不过是把硬件和软件插入联网的环境，并立即就能使用可用的服务：就像我们今天插接电话一样。在今天，当消费者从商店购买一部电话后，他不必对电话进行配置。消费者只需给电话服务供应商打一个电话，服务就会送上门。最后，消费者只需把电话插好，就能使用电话服务了，自主地联网。

4) 能够简化与网络的交互性

从消费者的角度看，消费者把可插接的设备和软件插入网络，就像今天插接一部电话一样简单。

从传统服务供应商的角度看，JINI 简化了 Services Delivery (服务提供)的管理。设备不但能向网络推出增值服务，而且还能提供设备的属性和功能。现在，服务供应商可以针对每台设备设计服务。当然，JINI 还将有可能打开一扇通向新的网络化服务的大门。

从产品制造商的角度看，JINI 打开了全新的市场。因为 JINI 简化了设备向网络提供增值服务的能力。所以，产品就不仅仅作为商品而投入竞争，而是作为增值服务的产品参与竞争。

从 Java 程序员的角度看，JINI 简化了编写分布式应用程序的工作，因而，任何 Java 程序员都能利用基于 JINI 的新设备编写应用程序和服务。因此，企业不再需要聘用有限的专家资源编写分布式应用程序，任何 Java 程序员都能为基于 JINI 的网络开发服务。

JINI 的起源 Bill Joy 在 1994 年之前向 Sun 公司实验室提交了一份包括以下三个主要概念的建议书：

可在所有平台上运行的语言、运行该语言的虚拟机和允许分布式虚拟机像单一系统那样工作的网络化系统。1995 年，这种语言和虚拟机相继面市，即 Java 编程语言和 Java 虚拟机。但该系统的概念则仍保留在 Sun 公司的研究与开发实验室，作进一步的研究和开发。这个系统的概念就是 JINI。

JINI 战略部署与合作伙伴 Sun 公司部署了广泛的战略，力求将 JINI 推向市场。我们可以这样说，JINI 与任何向网络化环境提供产品和/或服务的企业都密切相关。这包括传统的设备制造商、服务供应商和软件开发商。

4.5.4 JINI 的工作过程

一个完整的 JINI 系统由基础设施(Infrastructure)、编程模型(Programming Model)、服务(Service)三个部分组成，其体系结构如图 4-6 所示。

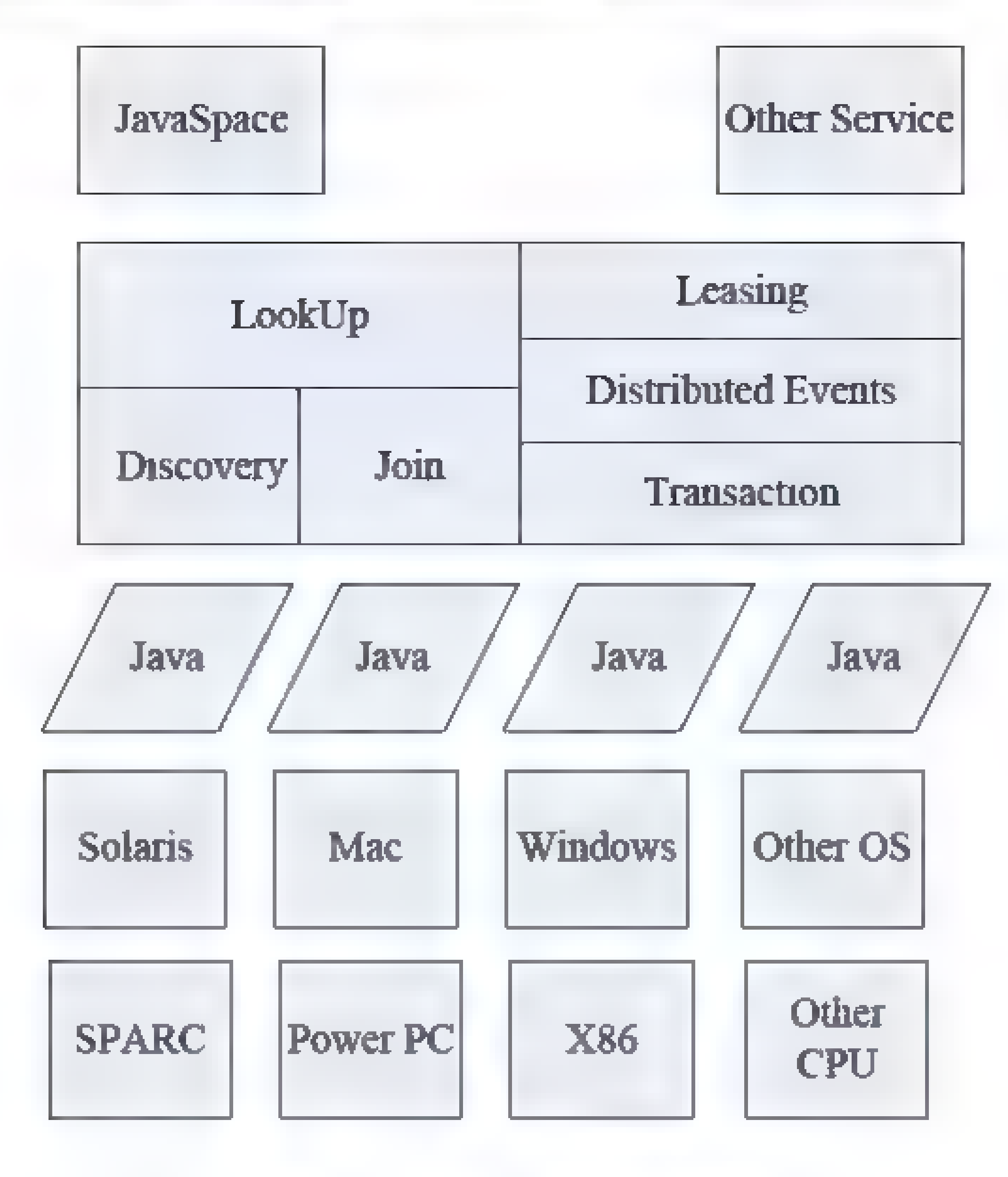


图 4-6　JINI 的体系结构

1. 基础设施

基础设施用来定义基于 JINI 的硬件设备和软件组件如何连接并注册到网络，它包括以下 4 个组成部分：

- Java RMI 扩展实现：是 JINI 系统中的构件在通信时所采用的底层通信机制。
- 分布式安全系统：用于将 Java 平台的安全模型扩展到分布式 JINI 系统，并定义联盟成员的使用权限。
- 发现 / 加入(Discovery/Join)协议：是一种服务协议，允许软硬件发现网络并成为联盟的成员，同时将所提供的服务广播给联盟中的其他成员。
- 查找(Lookup)服务：是网络中所有服务的公告牌(Bulletin Board)，用来展示联盟中的所有成员，并且帮助使用者在联盟中寻找所需的资源和服务。

JINI 基础设施负责实现添加、删除、定位和访问服务的相关机制，它通常驻留在网络中的三个地方：在查找服务中，在服务提供者中，或者在客户中。JINI 基础设施的核心是 Lookup、Discovery 和 Join 三条协议，它们使得基于 JINI 的任何服务都可以随时加入或者退出联盟，并且在加入联盟时无需进行安装和配置，从而达到了即插即用的效果。

查找服务(Lookup Service) 是 JINI 体系结构中的基本组成部分，它负责在分布式系统中提供对服务的中央注册机制。一旦进入 JINI 的世界，如果想找到所需的服务就必须通过查找服务，此外，查找服务还是为管理员和用户提供各种访问接口的基础。在一定程度上你可以将查找服务看成是网络中所有服务的公告牌，它维护着各个服务所提供的功能接口与实现该服务的对象集间的映射关系。JINI 中的客户通过使用查找服务在分布式网络中查找和调用所需的服务，而查找服务中的对象本身可以包含其他查找服务，从而构成层次式的查找服务。

当硬件设备或者应用程序进入网络时，它所提供的服务如何被 JINI 发现并接纳，并

进一步成为整个 JINI 分布式系统中的某项服务，需要经过两个必不可少的重要步骤。第一步是找到系统中的查找服务，这个过程称为发现(Discovery)，所用到的协议是发现协议。第二步是将自己注册到查找服务中，这个过程称为加入(Join)，描述这个过程的协议是 Join 协议。一个 JINI 服务只有在成功完成了这两个步骤之后，才能正式成为联盟中的一员，并开始向外界提供相应的服务。

在 JINI 这一分布式计算环境中，服务提供者所提供的服务既可能是硬件也可能是软件。当某一服务提供者需要加入到 JINI 系统中时，它首先必须找到网络中的查找服务，于是它在局域网中进行广播，请求加入到查找服务中。附近的查找服务在收到相应的请求后负责识别并接纳该服务，整个过程称之为发现。需要注意的是，服务提供者必须包含用于通讯的服务对象(Service Object)和描述该服务特点的服务属性(Service Attributes)。

当查找服务找到之后，服务提供者将与查找服务直接进行通讯，把自己的服务对象和服务属性注册到查找服务中，换句话说就是把服务对象和服务属性发送到查找服务中去。这个过程称为加入。

一旦服务被成功地加入到查找服务中后，如果 JINI 系统中的客户需要使用该服务，它可以根据服务的类型或者属性向查找服务查询合适的服务，查找服务负责把查询后的结果返回给客户，当客户决定使用该服务时，查找服务还会将该服务对象的拷贝发送给客户。

JINI 客户从查找服务那里获得的服务对象是一个 Java 接口(Interface)，其中包括用来调用服务的方法名称和参数，以及其他一些描述信息。JINI 客户通过获得的服务对象与服务提供者进行直接联系，获得相应的服务。服务对象负责处理客户与服务提供者之间的通信，从而向用户隐藏了服务的具体实现细节。

2. 编程模型

JINI 是一个分布式的计算环境，其编程模型自然也是分布式的，编程模型在 JINI 的体系结构中占有非常重要的地位，其基础设施正是借助这一编程模型有机地结合在了一起。JINI 的编程模型主要包括以下几个方面。

1) 租用(Leasing)

在分布式系统中有一个非常严重的问题，那就是不能保证服务不突然崩溃。例如，当一台数字相机通过查找服务加入到 JINI 网络中时，将对外发布信息表明自己可用且一切正常，但如果此时用户在不正常关闭设备的情况下随意将相机拔掉，相应的问题就产生了。因为对联盟中的其他成员来说，此时它们无法判断是相机所连接的远程主机已经关掉，还是响应速度比较慢，或是相机本身产生了故障。为了解决这个问题，JINI 使用了一种称为租用的技术。租用的基本思想是：不再保证可以在任意长的时间内访问资源，而是只能在一段固定时间内将资源“借给”某使用者。租用机制使得客户对服务的访问是基于租约的，租约保证了一段时间内的授权访问，租约必须在服务的使用者和服务提供者之间进行协商。租约在到期之前如果不续约的话，相应的资源将被释放。租约可以是唯一的，也可以不是唯一的，非唯一租约允许多用户共享同一资源。

2) 分布式事件(Distributed Events)

和 Java 类似，JINI 也使用事件的概念来处理异步通知，事件可以被理解成是一个包

含外部状态变化信息的对象。例如，当鼠标状态在 AWT 中发生变化时，不论是移动、点击还是释放，都将产生一个 MouseEvent 事件，事件产生后将被直接发送到希望得到该信息的相关实体(Listener)。JINI 的事件模型和 Java 的非常类似，但它同时还能够支持分布式事件。JINI 的编程模型允许一个对象注册对其他对象感兴趣的事件，并且在该事件发生时能立即得到相应的通知，这就使得基于事件的分布式程序能以有多种可靠性和扩展性保证的方式编写。

3) 事务 (Transaction)

分布式计算中另一个非常棘手的问题是部分失败，所谓部分失败指的是整个计算过程中的某一步产生了错误，或者计算所需的某个组成部分出现了故障。假如，银行想在两个账户间进行转账，即从一个账户中减去一定数量的金额然后加到另一个账户上，是不是只要编写一段代码实现从 A 账户中减去一定的金额，然后再用另外一段代码实现向 B 账户中加入同样的金额就可以了呢？看起来似乎没有什么问题，但实际上却存在非常严重的错误，如果资金从账户 A 转出之后机器就崩溃了，很明显 B 将永远无法得到，而产生这一问题的根本原因就是没有考虑到程序可能会部分失败。JINI 通过引入事务的概念解决了部分失败这一问题，事务是将一系列相关操作进行分组，以保证所有操作要么全部成功，要么全部失败的一种方法。采用事务的好处是无论在执行过程中出现什么情况，系统都将进入一个确定的状态，这样处理起来就相对容易多了：事务成功则继续进行，事务失败则稍后再试。事务最早出现在数据库理论中，目前已经出现了许多实用的事务模型，JINI 中采用的是两阶段提交(Two-phase Commit)这一事务模型。

3．服务

JINI 的基础设施和编程模式使得服务能够在分布式环境中被注册和发现，并能够向用户宣布自己的存在。服务是 JINI 体系结构中一个非常重要的概念，它可以用来表示组织在一起形成 JINI 联盟的各个实体，这里所指的实体可以是硬件、软件或者软硬件的结合。

JINI 中的每个服务都有一个接口描述，该接口定义了客户可以向这个服务请求的所有操作，并且反映了服务的类型。JINI 中的服务是可以聚合的，即允许一个服务由多个子服务组合而成。整个 JINI 体系结构中最重要的一个服务是查找服务，它是 JINI 基础设施的一个子组件，其他作为 JINI 体系结构组成部分并实现为 JINI 服务的对象包括：

(1) JavaSpace：为 JINI 中的对象提供了一个可选的分布式持续性保存机制，并能够被用来进行简单的通信。

(2) 事务管理器：为 JINI 中的对象提供分布式事务服务，允许对象参与到由编程模式所定义的两阶段提交协议。

4.5.5 服务对象的优点

在 JINI 系统中，网络移动性对象可以移动到任何地方。当客户机或者服务进行探索的时候，它会从查找服务受到一个服务注册器。当通过服务注册器进行加入的时候，它会传送一个服务条目对象给查找服务，而服务条目本身也是一个包含了很多对象的容器，其中

包括属性和服务对象。当客户进行查找的时候，它会发送一个服务模板对象，包含一系列对象，表示查询需要的搜索条件。如果查询成功了，客户机会收到匹配查询的服务对象，或者是服务的整个服务条目。

这些在网络上客户机、服务和查找服务之间移动的对象到底能对分布式程序有什么好处呢？简单说来，JINI 使用的网络移动的对象(特别是网络移动的服务对象)提高了分布式系统编程的抽象级别，有效地把网络编程转变为面向对象编程。

JINI 体系通过把面向对象编程引入网络，带来了面向对象的一个基本有点：接口和实现分离。比如，服务对象允许客户机用好几种方法来获取服务，客户从查询服务下载、在本地运行的服务对象可以代表整个服务。或者说，服务对象可以作为远程服务器的一个代理。当客户机调用服务对象的方法的时候，它把请求通过网络传送回服务器，那才是真正工作的地方。本地服务对象和远程服务器也可能共同分担工作。

JINI 体系的一个重要推论就是，在服务对象代理和远程服务器之间使用的网络协议，客户机是无需关心的额。网络协议是实现服务的一部分，协议完全是服务开发者的私人事务。客户机可以通过这种私有协议和服务器交互，因为服务把它自己的服务对象送到客户机的地址空间中——服务对象在服务和客户机的网络上来回传送。注入客户机的服务对象可以用任何协议和后端的服务通信，RMI、CORBA、DCOM，或者是自己在 socket 和流上面建立的协议，甚至是其他的任何方法。客户端完全不需要关心网络协议，因为它只需和服务对象实现的公开接口打交道。服务对象负责人和需要进行的网络交流。

同一服务接口的不同实现可以使用完全不同的方法和完全不同的网络协议。服务可能使用特制的硬件来满足客户机的请求，或者使用软件实现所有功能。服务不同的实现方法可以针对不同的环境优化。另外，服务采用的方法可能随时间而变化，客户机可以确信服务对象了解服务是如何实现的，因为客户机就是从服务那儿取得这个服务对象的。对于客户机来说，不管服务是如何实现的，任何服务看起来都是公开的接口。

因此，JINI 试图提升分布式系统编程的抽象级别，从网络协议级别提升到对象接口级别，在越来越多嵌入式设备都要连接到网络上的情况下，分布式系统的各个部分可能来自不同的供应商。JINI 让供应商不需要依附于某一种底层网络协议(这种协议让它们的设备互联)。相反，供应商只需要在互联设备的高层 Java 接口层上达成一致就可以了。讨论的级别从网络协议层提升到对象接口层，可以让供应商更加集中于高层的概念而非拘泥在基层细节中。JINI 讨论的高层问题可以使类似产品的供应商达成一个一致协议，来描述它们的服务如何和客户机交互。

除此之外，JINI 体系允许软件开发者在开发分布式系统时享受接口和实现分离的便利。一个好处是良好设计的对象接口可以使软件开发者在大规模的分布式系统项目中更加有效地协同开发。对象接口为面向对象程序的各个部分之间签订了合同，同样，对象接口也可以用来明确大项目中团队成员之间的互相合作关系，他们每人负责一块程序。接口和实现分离的另外一个好处就是，程序员可以用它来减少变化带来的冲击，因为耦合度降低了。设计良好的对象直接结合的唯一途径就是它们的接口，实现者在对象内部做的改变不会影响到其他对象中的代码。

通过为分布式系统编程提升抽象层次和提供清晰的分离接口和实现，JINI 带来了面向

对象的好处。这都是因为 Java 对网络移动对象的支持。对象在网络上移动，是通过 Java 的底层结构，对象序列化，RMI 来实现的，并通过 JINI 服务对象展示出来，这些技术给网络带来了巨大的好处。

4.5.6 JINI 技术的运作

可以将 JINI 技术划分为两个范畴：体系结构和分布式编程。此外，还将提供在 JINI 上运行的网络服务。

1) 基础结构

JINI 基础结构解决设备和软件如何与网络连接并进行注册等基本问题。基础结构的第一种要素称作 Discovery and Join (发现与联合)。Discovery and Join 解决设备和应用程序在对网络一无所知的情况下如何向网络进行首次注册这样的难题。

基础结构的第二个要素是 Lookup(搜索)，Lookup 可以被看作是网络中所有服务的公告板，各个搜索元素的具体说明如下。

- Network Services：网络服务。
- Other Services：其他服务。
- Leasing：租用。
- Transactions：交易。
- Distributed Event：分布式事件。
- Other OS：其他操作系统。
- Other CPU：其他 CPU。
- DISCOVERY AND JOIN：设备或应用程序插入网络后需要完成的第一个任务就是发现该网络，并使网络发现该设备或应用程序。我们之所以使用 Discovery and Join 这样的说法，是因为设备或应用程序事前不可能对网络有任何了解。

Discovery 的工作原理：当基于 JINI 的设备插入网络后，它就通过一个众所周知的端口向网络发送一个 512 字节的多路广播 Discovery 包。在其他信息中，该包包含对自己的引用。

JINI Lookup 在众所周知的端口上进行监听。当接收到 Discovery 包后，Lookup 就利用该设备的接口将 Lookup 的接口传递回插接的设备或应用程序。

现在，该设备或应用程序已经发现了该网络，并准备将其所有特性上载到 JINI Lookup。上载特性是 Discovery and Join 中 Join 这方面的特性。

现在该设备或应用程序使用在 Discovery 阶段所接收到的 Lookup 接口与网络相连。上载到 Lookup 的特性包括该设备或应用程序所提供的所有增值服务(如驱动程序、帮助向导、属性等)。

LookupLookup 是网络上所有服务的网络公告板。Lookup 不但存储着指向网络上服务的指针，而且还存储着这些服务的代码或代码指针。

例如，当打印机向 Lookup 注册时，打印机将打印机驱动程序或驱动程序接口上载到 Lookup。当客户机需要使用打印机时，该驱动程序和驱动程序接口就会从 Lookup 下载到客户机。这样，就不必事先把驱动程序装载到客户机上。

打印机还可能把其他增值服务装载入 Lookup。例如，打印机可能存储关于自己的属性(如它是否支持 postscript，或它是否为彩色打印机)。打印机还可能存储可在客户机上运行的帮助向导。

如果网络上没有 Lookup，则网络就会使用一个 Peer Lookup (对等 Lookup)程序。当需要服务的客户机在网络上找不到 Lookup 时，Peer Lookup 就开始工作。在这种情况下，客户机可发送与 Lookup 所用的相同的 Discovery and Join 包，并要求任何服务供应商进行注册。随后，服务供应商就会在客户机上注册，尽管那不是 Lookup。分布式编程 JINI 分布式编程为 Java 增添了创建分布式系统所必需的其他功能。尤其是 JINI 分布式编程可提供租用、分布式交易和分布式事件。

在 JINI 中，对象彼此之间商定租期。例如，当某设备使用 Discovery and Join 协议发现网络时，它就注册一段租用时间。在租约到期之前，该设备必须重新商定租期。这样，如果租约到期或设备拔下后，该设备在 Lookup 中的记录就会被自动删除。这就是分布式垃圾收集的工作原理。

2) 分布式的影响

分布式事件在单一的计算机中，事件肯定能被接收方接收到，序列也肯定能按照顺序进行。但是在分布式环境中，分布的事件可能不是按照顺序被接收，或者，某个事件还可能丢失。

为了便于在 Java 环境中处理分布的事件，JINI 为分布的事件提供了一个简单的 Java API。例如，当一个分布的事件发生时，该事件都带有一个事件号和序列号。利用这种信息，接收方就能检查事件是否丢失(序列号丢失)或事件是否按照顺序接收(序列号顺序不对)到。

分布式交易在分布式 Java 环境中，有时需要一种很简便的方法，来确保在整个交易完成之前，在该交易中发生的所有事件都被真正提交了(两阶段提交)。

为便于进行此类分布式计算，JINI 提供了一种简单的 Java API。该 API 可使对象启动一个能管理交易的交易管理器。每个参与交易的对象都向交易管理器注册。

当发生交易时，如果某个参与的对象说，交易中的某个事件没有发生，则此信息就被送回交易管理器。随后，交易管理器就告诉所有参与的对象回滚(Rool Back)到前一个已知状态。类似的，如果所有对象都完成了其交易的过程，则整个交易就向前进行。

JINI 上的网络服务在 JINI 基础结构和分布式编程之上，可提供便于分布式计算的网络服务。JavaSpace 就是这样的一种网络服务。

4.5.7 如何启动 JINI

JINI 目前还处于研发的早期阶段，因此迄今为止还没有哪个 JINI 实现系统声称它可以真正投入实际使用，自然也就没有完全遵循 JINI 规范的服务或者设备出现。尽管如此，JINI 试图去解决的问题对许多软硬件厂商来讲已经非常急切了，因此 Java 给出了一个 JINI 的参考实现系统 JINI Technology Starter Kit，读者可以从 http://www.sun.com/software /JINI/上下载，目前最为常用的版本是 2.0。

从 Java 官方网站下载到 JINI Technology Starter Kit Version 2.0 的源码包 JINI-2 0-

src.zip 后，假设将其解压缩到 d:\JINI2_0\目录下。由于 JINI 建立在一系列其他服务的基础之上，因此在正式启动 JINI 之前还必须先对这些服务进行配置，JINI Technology Starter Kit 开发包中集成了一些支持 JINI 运行所必需的基本服务。

1) HTTP 服务器

在通过 RMI 下载代码时，需要利用 HTTP 协议完成传输，因此在运行 JINI 时需要一个 HTTP 服务器。JINI Technology Starter Kit 带有一个非常简单的 HTTP 服务器，但它对于供在应用程序间传送代码已经完全足够了。通常情况下，需要在每个供其他应用程序下载代码的机器上运行一个 HTTP 服务器。

2) RMI 激活守护进程

这是 JINI Technology Starter Kit 基础结构中非常实用的一个部分，它使得那些很少被调用的对象基本保持在“睡眠”状态，在需要时又能够被自动“唤醒”。RMI 激活守护进程负责管理对象在活跃和非活跃状态间的切换，并被其他 JINI 运行时核心服务所调用，它至少应该在查找服务所处的主机上运行。

3) 查找服务

查找服务才是 JINI 的核心，它负责记录网络中当前激活的所有服务。尽管 JDK 中的 RMI 注册服务也可以作为查找服务使用，但不推荐使用，JINI Technology Starter Kit 中提供的查找服务具有更加丰富和完善的功能。当出现故障或重新启动之后，查找服务需要依靠 RMI 激活守护进程来恢复其状态，因此在运行查找服务的机器上必须同时运行 RMI 激活守护进程。

JINI Technology Starter Kit v2.0 在它的 tools.jar 包中自带了一个 HTTP 服务器，我们可以在 d:\JINI2_0\script\目录下创建一个内容如下的批处理文件来启动它。

代码清单 1　start-httpd.bat

```
@rem 启动 HTTP 服务器的批处理文件
java -jar lib\tools.jar -port 8080 -dir lib -verbose
```

在缺省情况下，HTTP 服务器将运行在 8080 端口，但如果此时系统中已经有一个 Web 服务器运行在 8080 端口了，那么可以用-port 参数把 JINI 的 HTTP 服务器指定到另外的端口。参数-dir 可以用来设置 HTTP 服务器的根目录，在上面的例子中将其设置为 JINI 的 lib 目录，这样做的好处是 JINI 的核心代码可以被直接下载。最后那个参数-verbose 是用于调试的，它要求 HTTP 服务器显示所有来自客户端的请求信息以及这些请求的来源。

RMI 激活守护进程 rmid 是由 JDK 所提供的，它必须在可激活对象所在的每一台主机上运行，其中包括 JINI 查找服务、事务管理器和 JavaSpace。如果你已经成功地配置好了 JDK，那么可以在 d:\JINI2_0\script\目录下创建一个内容如下的批处理文件来启动它：

代码清单 2　start-rmid.bat

```
@rem 启动 RMI 激活守护进程的批处理文件
rmid -J-Dsun.rmi.activation.execPolicy=none
```

在 JINI Technology Starter Kit v2.0 中是通过 Reggie 实现查找服务的，它位于 reggie.jar 包中，启动这个服务比启动其他服务要稍微复杂一些。首先在 d:\JINI2_0\config\目录下为

Reggie 创建一个内容如下的安全策略文件：

代码清单 3　all.policy

```
/* 安全策略文件 */
grant codeBase "file:lib${/}*" {
  permission java.security.AllPermission;
};
```

然后在 d:\JINI2_0\config\目录下为 Reggie 创建一个内容如下的启动配置文件：

代码清单 4　start-reggie.config

```
/* 启动 Reggie 查找服务的配置文件 */
import com.sun.JINI.config.ConfigUtil;
import com.sun.JINI.start.NonActivatableServiceDescriptor;
import com.sun.JINI.start.ServiceDescriptor;
com.sun.JINI.start {
  private static codebase =
     ConfigUtil.concat(
        new Object[] {
           "http://",
           ConfigUtil.getHostName(),
           ":8080/reggie-dl.jar" });
  private static policy = "config${/}all.policy";
  private static classpath = "lib${/}reggie.jar";
  private static config = "config${/}reggie.config";
  static serviceDescriptors =
     new ServiceDescriptor[] {
         new NonActivatableServiceDescriptor(
           codebase,
           policy,
           classpath,
           "com.sun.JINI.reggie.TransientRegistrarImpl",
           new String[] { config })
  };
}
```

最后再在 d:\JINI2_0\config\目录下为 Reggie 创建一个内如下容的配置文件：

代码清单 5　reggie.config

```
/* Reggie 查找服务的配置文件 */
import net.JINI.jrmp.JrmpExporter;
com.sun.JINI.reggie {
  serverExporter = new JrmpExporter();
  initialMemberGroups = new String[] { "example.JINI.sun.com" };
}
```

为了简化 Reggie 查找服务的启动过程，我们可以在 d:\JINI2_0\script\目录下创建一个如下内容的批处理文件：

代码清单 6　start-reggie.bat

```
@rem 启动 Reggie 查找服务的批处理文件
java -Djava.security.policy=config\all.policy
   -jar lib\start.jar config\start-reggie.config
```

这样在需要启动 HTTP 服务器、RMI 激活守护进程或者 Reggie 查找服务时，只用在命令行方式下依次执行如下命令即可：

```
D:\JINI2_0> start script\start-httpd
D:\JINI2_0> start script\start-rmid
D:\JINI2_0> start script\start-reggie
```

在 JINI 的世界中，所有可用的资源都被看成是服务，而对服务的查询和检索则是最频繁的操作之一，为此 JINI Technology Starter Kit v2.0 专用提供了一个实用工具 Browser，利用它可以查询当前 JINI 网络中的所有服务。为了简化 Browser 的启动，可以在“d:\JINI2_0\script\”目录下创建一个内容如下的批处理文件：

代码清单 7 start-browser.bat

```
    @rem 启动 Browser 的批处理文件
    java -Djava.security.policy=config\all.policy -
Djava.rmi.server.codebase=
    http://%computername%:8080/browser-dl.jar -jar lib\browser.jar
```

此时要想启动 Browser，只要在命令行方式下依次执行如下命令即可。

```
D:\JINI2_0> start script\start-browser
```

正如前面介绍过的，查找服务是 JINI 中的一项特殊服务，因而此时如果 Reggie 查找服务已经正常启动了，那么可以在 Browser 中查询到它，如图 4-7 所示。为了监视 JINI 查找服务的运行状态，可以让 Browser 一直运行下去。

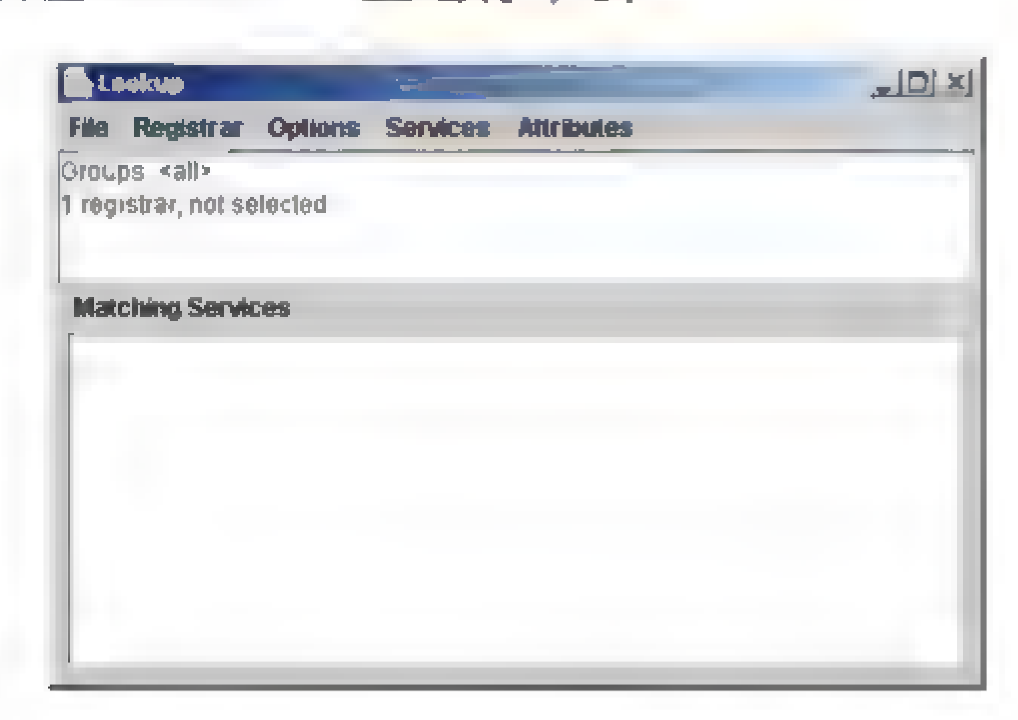


图 4-7　Browser 查询界面



第5章

浅谈 Java 虚拟机的内部机制

前面已经讲解了整个 Java 技术的体系结构，并且也介绍了 Java 虚拟机在 Java 技术体系中相对于其他组成部分所扮演的角色。从本章开始，将正式步入 Java 虚拟机的知识，本章首先对 Java 虚拟机的内部机制进行概览性介绍，为读者学习后面的知识打下基础。

5.1 什么是虚拟机

Java 虚拟机之所以被称之为是“虚拟”的，就是因为它仅仅是由一个规范来定义的抽象计算机。因此，要运行某个 Java 程序，首先需要一个符合该规范的具体实现。本章主要描述这个规范本身，但是为了更细致地描述某些具体特性，我们也将讨论它们可能以哪些方式来实现。本章将详细讲解 Java 如何把握网络所带来的巨大机遇，为学习本书后面的知识打下基础。

要理解 Java 虚拟机，首先必须意识到，当你说“Java 虚拟机”时，可能指的是以下三种不同的东西：

- 抽象规范。
- 一个具体的实现。
- 一个运行中的虚拟机实例。

Java 虚拟机抽象规范仅仅是个概念，而该规范的具体实现，可能来自多个提供商，并存在于多个平台上。它或者完全用软件实现，或者以硬件和软件相结合的方式来实现。当运行一个 Java 程序的同时，也就运行了一个 Java 虚拟机实例。

5.1.1 JVM 简介

JVM 全称是 Java Virtual Machine，即 Java 虚拟机，也就是在计算机上再虚拟一个计算机。JVM 和我们使用 VMWare 不一样，那个虚拟的东西是可以看到的，这个 JVM 是看不到的，它存在内存中。我们知道计算机的基本构成是运算器、控制器、存储器、输入和输出设备，而这个 JVM 也有这成套的元素。运算器当然还是交给硬件 CPU 处理了，只是为了适应“一次编译，随处运行”的情况，需要做一个翻译动作，于是就用了 JVM 自己的命令集，这与汇编的命令集有点类似，每一种汇编命令集针对一个系列的 CPU，比如 8086 系列的汇编也可以用在 8088 上，但是就不能跑在 8051 上，而 JVM 的命令集则可以到处运行，因为 JVM 做了翻译，根据不同的 CPU，翻译成不同的机器语言。

JVM 中最需要深入理解的就是它的存储部分，JVM 是一个内存中的虚拟机，那它的存储部分就是内存了，我们写的所有类、常量、变量、方法都在内存中，这决定着我们的程序运行得是否健壮、是否高效。

5.1.2 JVM 的组成部分

JVM 虚拟机的运作结构如图 5-1 所示。

从图 5-1 中可以看到，JVM 是运行在操作系统上的，它与硬件没有直接的交互。我们再来看下 JVM 有哪些组成部分，如图 5-2 所示。

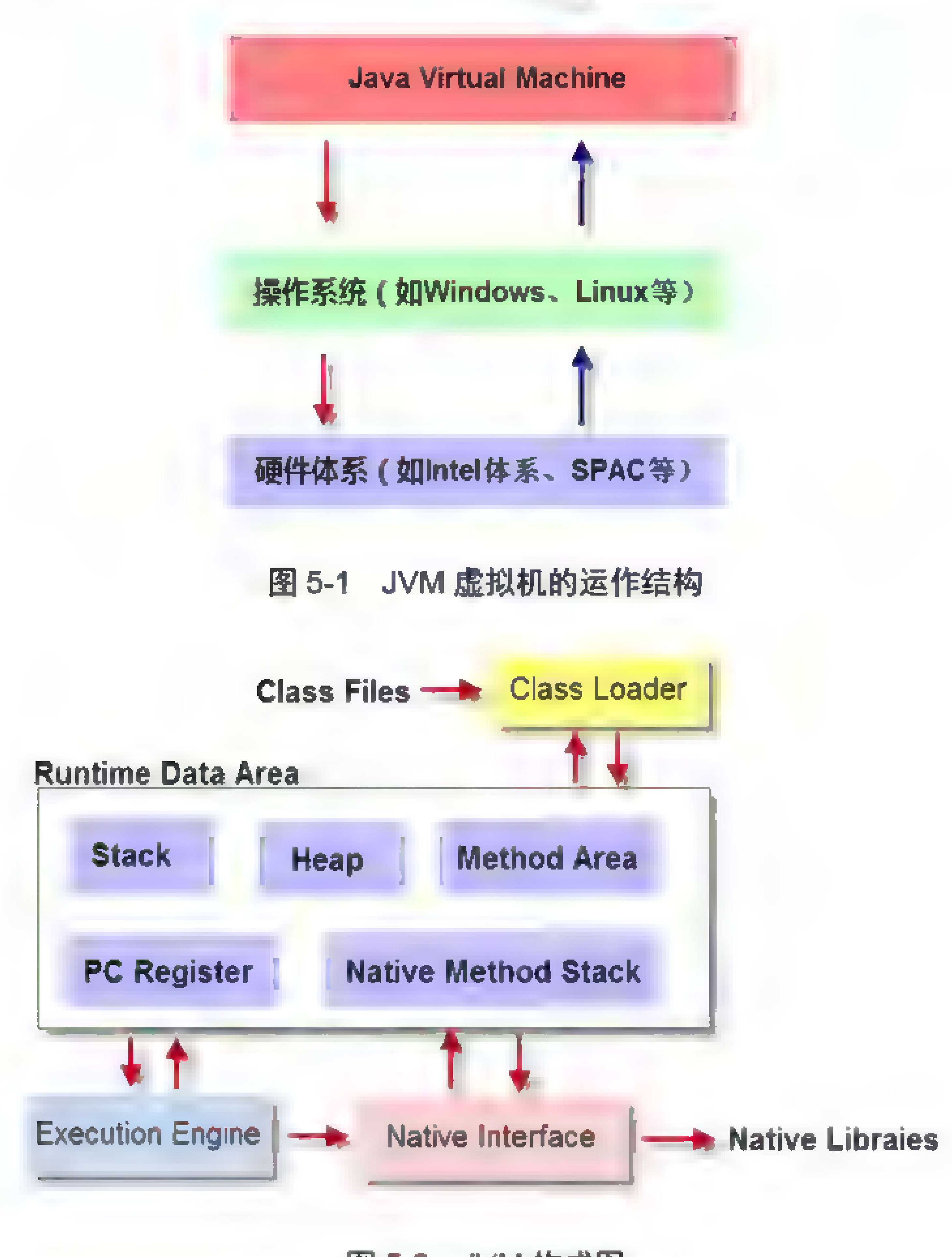


图 5-1 JVM 虚拟机的运作结构

图 5-2 JVM 构成图

5.2 Java 虚拟机的生命周期

一个运行时的 Java 虚拟机实例的天职是负责运行一个 Java 程序。在启动一个 Java 程序的同时会诞生一个虚拟机实例，当该程序退出时，虚拟机实例也随之消亡。如果在同一台计算机上同时运行三个 Java 程序，会得到三个 Java 虚拟机实例。每个 Java 程序都运行于它自己的 Java 虚拟机实例中。

Java 虚拟机实例通过调用某个初始类的 main()方法来运行一个 Java 程序。而这个 main()方法必须是公有的(Public)、静态的(Static)并且返回值为 void，还要接受一个字符串数组作为参数。任何拥有这样一个 main()方法的类都可以作为 Java 程序运行的起点。假如存在这样一个 Java 程序，此程序能够打印出传给它的命令行参数：

```
package jvm.ext1;
public class Echo {
    public static void main(String[]args) {
```

```
            int length = args.length;
            for (int i = 0; i <length; i++) {
                System.out.print(args[i] +"");
            }
            System.out.println();
        }
}
```

上述代码必须告诉 Java 虚拟机要运行的 Java 程序中初始类的名字，整个程序将从它的 main()方法开始运行。现实中一个 Java 虚拟机实现的例子如 SunJava 2 SDK 的 Java 程序。比如，如果想要在 Windows 上使用 Java 来运行 Echo 程序，需要输入如下命令：

```
java Echo Greetings, Planet
```

该命令的第一个单词“java”，告诉操作系统应该运行来自 Sun Java 2 SDK 的 Java 虚拟机。第二个词“Echo”则支持初始类的名字。Echo 这个初始类中必须有个公有的、静态的方法 main()，它获得一个字符串数组参数并且返回 void。上述命令行中剩下的单词序列“Greeting，Planet”作为该程序的命令行参数以字符串数组的形式传递给 main()，因此，对于上面这个例子，传递给类 Echo 中 main()方法的字符串数组参数的内容就是：

```
args[0]为"Greeting,"
args[1]为"Planet."
```

Java 程序初始类中的 main()方法，将作为该程序初始线程的起点，任何其他的线程都是由这个初始线程启动的。

在 Java 虚拟机内部有两种线程：守护线程与非守护线程。守护线程通常是由虚拟机自己使用的，比如执行垃圾收集任务的线程。但是，Java 程序也可以把它创建的任何线程标记为守护线程。而 Java 程序中的初始线程——就是开始与 main()的那个，就是非守护线程。

只要还有任何非守护线程在运行，那么这个 Java 程序也在继续运行(虚拟机仍然存活)。当该程序中所有的非守护线程都终止时，虚拟机实例将自动退出。假若安全管理器允许，程序本身也能够通过调用 Runtime 类或者 System 类的 exit 方法来退出。

在上面的 Echo 程序中，方法 main()并没有调用其他的线程。所以当它打印完命令行参数后返回 main()方法。这就终止了该程序中唯一的非守护线程，最终导致虚拟机实例退出。

5.3 Java 虚拟机的体系结构

在 Java 虚拟机规范中，一个虚拟机实例的行为是分别按照子系统、内存区、数据类型以及指令这几个术语来描述的。这些组成部分一起展示了抽象的虚拟机的内部抽象体系结构。但是规范中对它们的定义并非要强制规定 Java 虚拟机实现内部的体系结构，更多的是为了严格地定义这些实现的外部特征。规范本身通过定义这些抽象的组成部分以及它们之间的交互，来定义任何 Java 虚拟机实现都必须遵守的行为。

图 5-3 是 Java 虚拟机的结构图，包括在规范中描述的主要子系统和内存区。第 4 章我们曾提到，每个 Java 虚拟机都有一个类装载器子系统，它根据给定的全限定名类装入类型(类或接口)，同样，每个 Java 虚拟机都有一个执行引擎，它负责执行那些包含在被装载类的方法中的指令。

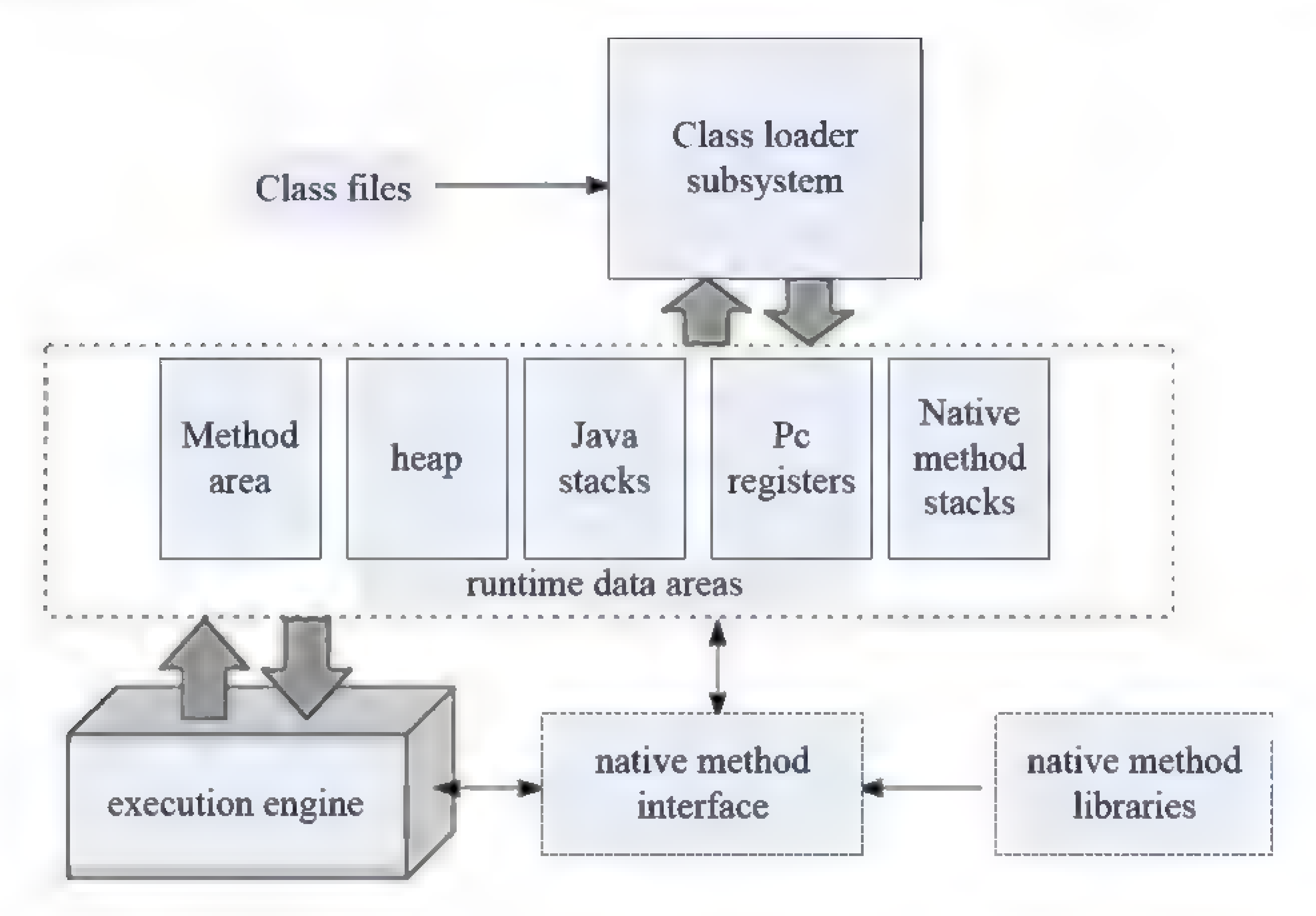


图 5-3　Java 虚拟机的内部体系结构

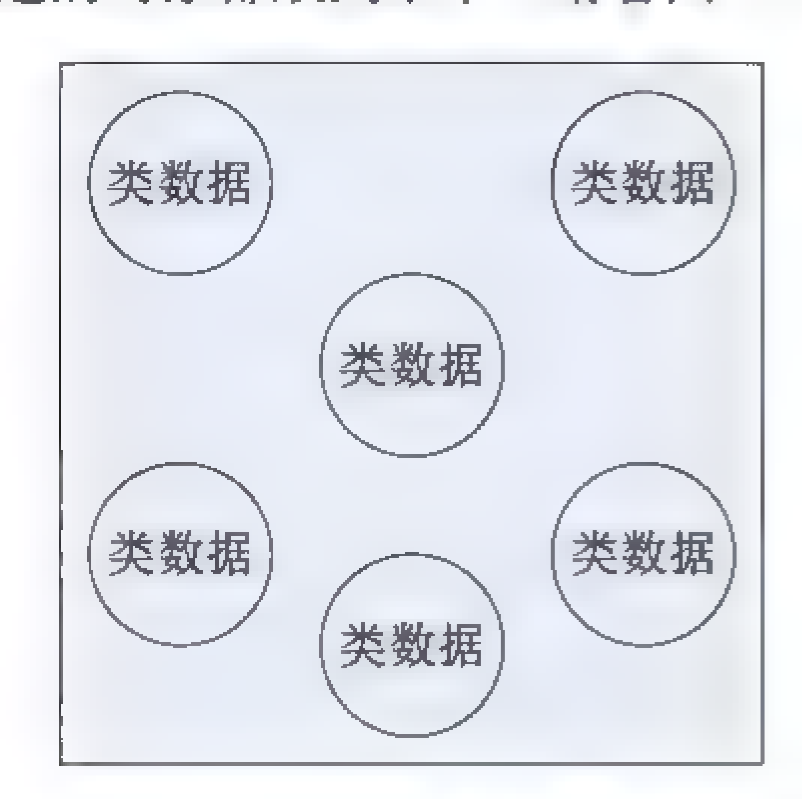
当 Java 虚拟机运行一个程序时，它需要使用内存来存储许多东西，例如下面的元素。

- 字节码。
- 从已装载的 class 文件中得到的其他信息。
- 程序创建的对象。
- 传递给方法的参数。
- 返回值。
- 局部变量。
- 运算的中间结果。

Java 虚拟机会把上述元素都组织到几个“运行时数据区”中，目的是便于管理。尽管这些“运行时数据区”都会以某种形式存在于每一个 Java 虚拟机实现中，但是规范对它们的描述却是相当抽象的。这些运行时数据区结构上的细节，大多数都由具体实现的设计者决定。

不同的虚拟机实现可能具有很不同的内存限制，有的实现可能有大量的内存可用，有的可能只有很少。有的实现可以利用虚拟内存，有的则不能。规范本身对“运行时数据区”只有抽象的描述，这就使得 Java 虚拟机可以很容易地在各种计算机和设备上实现。

某些运行时数据区是由程序汇总所有线程共享的，还有一些则只由一个线程拥有。每个 Java 虚拟机实例都有一个方法区以及一个堆，它们是由该虚拟机实例中所有线程共享的。当虚拟机装载一个 class 文件时，它会从这个 class 文件包含的二进制数据中解析类型信息。然后，它把这些类型信息放到方法区中。当程序运行时，虚拟机会把所有该程序在运行时创建的对象都放到堆中。请看图 5-4 中对这些内存区域的描绘。



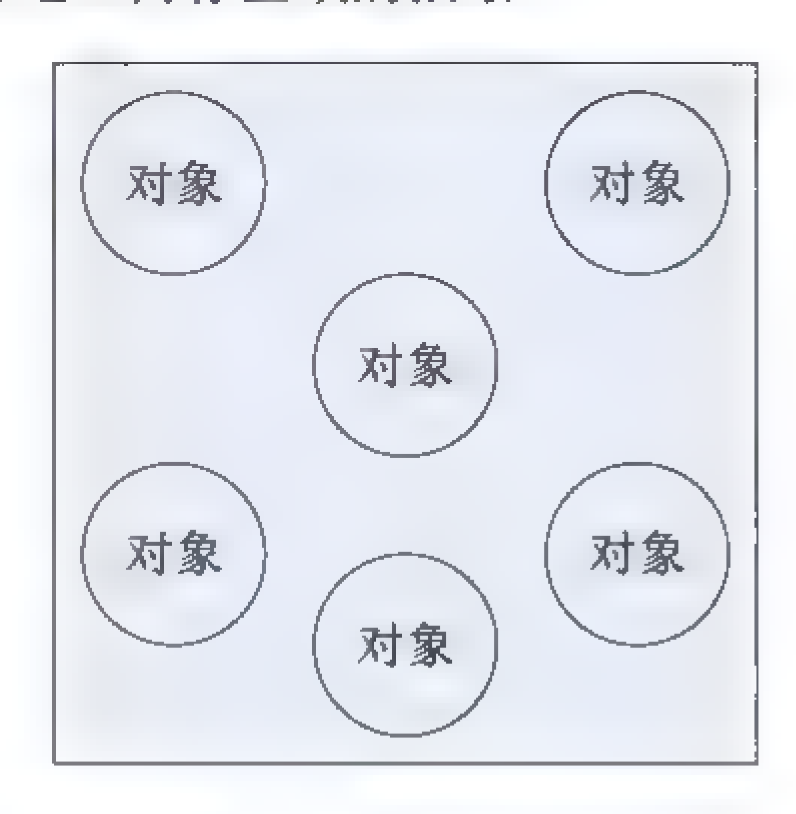


图 5-4　由所有线程共享的运行时数据区

当每一个新线程被创建时，它都将得到它自己的 PC 寄存器(程序计数器)以及一个 Java 栈：如果线程正在执行的是一个 Java 方法(非本地方法)，那么 PC 寄存器的值将总是指示下一条将被执行的指令，而它的 Java 栈则总是存储该线程中 Java 方法调用的状态——包括它的局部变量，被调用时传进来的参数，它的返回值，以及运算的中间结果等等。而本地方法调用的状态，则是以某种依赖于具体实现的方式存储在本地方法栈中，也可能是在寄存器或者其他某些与特定实现相关的内存区中。

Java 栈是由许多栈帧(Stackframe)或者说帧(Frame)组成的，一个栈帧包含一个 Java 方法调用的状态。当线程调用一个 Java 方法时，虚拟机压入一个新的栈帧到该线程的 Java 栈中：当该方法返回时，这个栈帧被从 Java 栈中弹出并抛弃。

Java 虚拟机没有寄存器，其指令集使用 Java 栈来存储中间数据。这样设计的原因是为了保持 Java 虚拟机的指令集尽量紧凑，同时也便于 Java 虚拟机在那些只有很少通用寄存器的平台上实现，另外 Java 虚拟机的这种基于栈的体系结构，也有助于运行时某些虚拟机实现的动态编译器和即时编译器的代码优化。

图 5-5 描绘了 Java 虚拟机为每一个线程创建的内存区，这些内存区域是私有的，任何线程都不能访问另一个线程的 PC 寄存器或者 Java 栈。

图 5-5 展示了一个虚拟机实例的快照，它有三个线程正在执行。线程 1 和线程 2 都正在执行 Java 方法，而线程 3 则正在执行一个本地方法。在图 5-5 中，和本书其他地方一样，Java 栈都是向下生长的，而栈顶都显示在图的底部，当前正在执行的方法的栈帧则以浅色表示，对于一个正在运行 Java 方法的线程而言，它的 PC 寄存器总是指向下一条将被执行的指令。在图 5-5 中，像这样的 PC 寄存器(比如线程 1 和线程 2 的)都是以浅色显示的。由于线程 3 当前正在执行一个本地方法，因此，它的 PC 寄存器——以深色显示的那个，其值是不确定的。

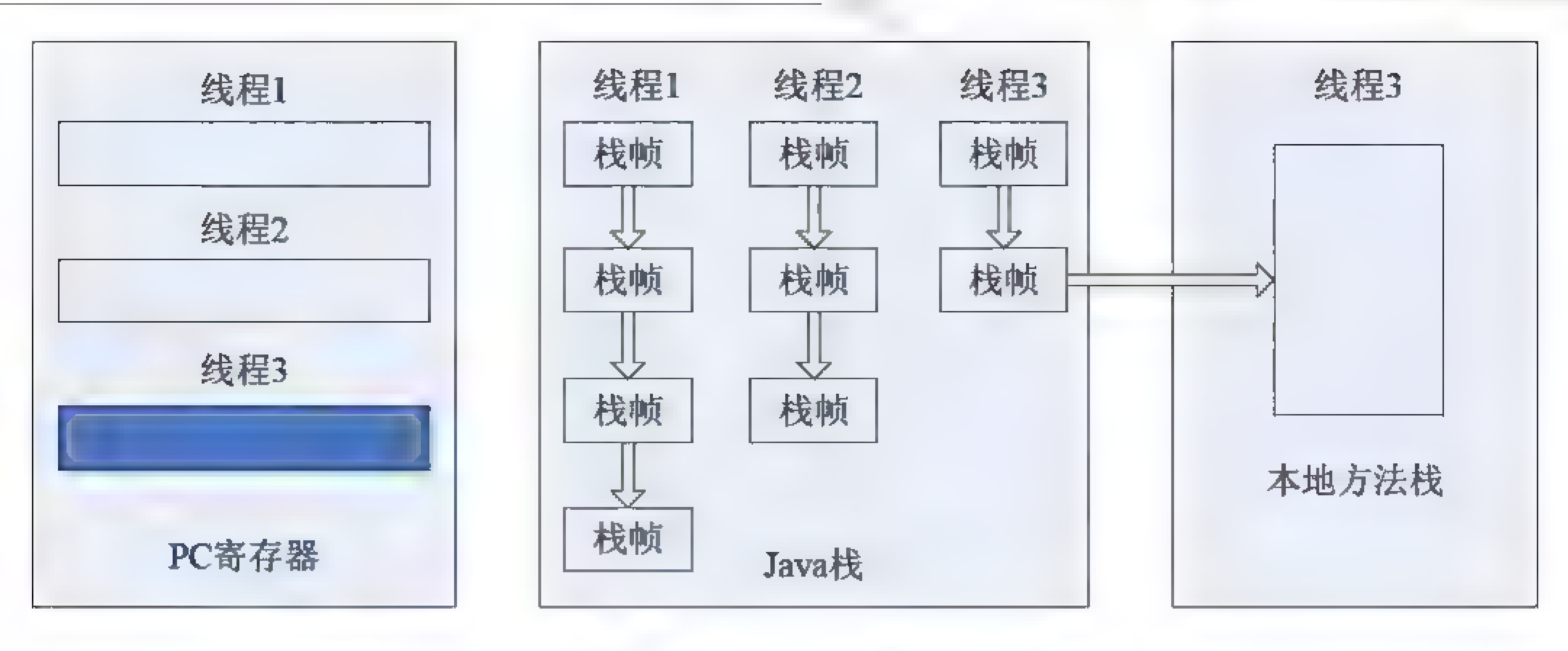


图 5-5 线程专有的运行时数据区

5.3.1 数据类型

Java 虚拟机是通过某些数据类型来执行计算的，数据类型及其运算都是由 Java 虚拟机规范严格定义的，数据类型可以分为两种：基本类型和引用类型。基本类型的变量持有原始值。而引用类型的变量持有引用值。术语“引用值”指的是对某个对象的引用，而不是该对象本身，与此相对，原始值则是真正的原始数据。请看图 5-6 中对 Java 虚拟机中数据类型的描绘。

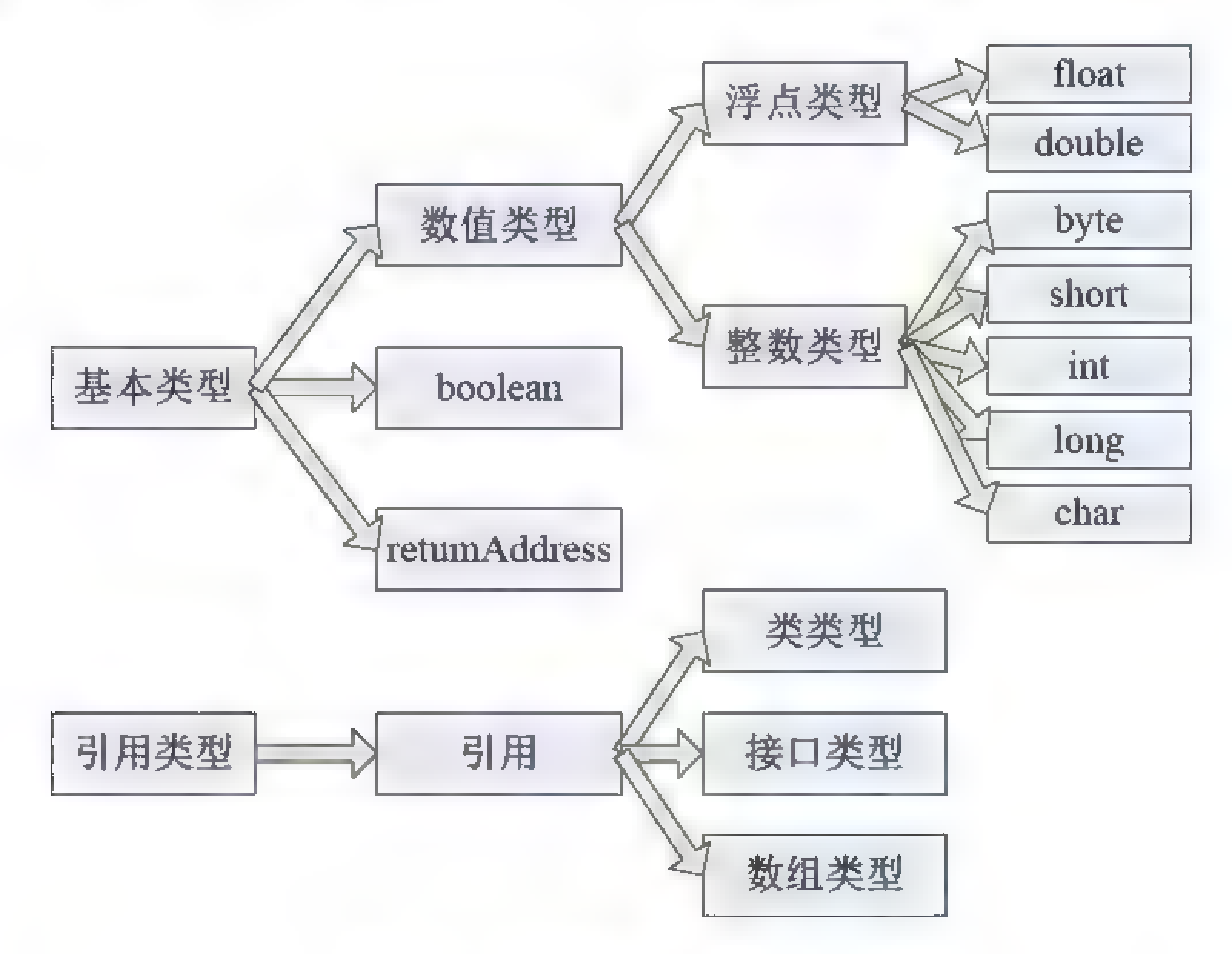


图 5-6 Java 虚拟机中的数据类型

Java 语言中的所有基本类型也都是 Java 虚拟机中的基本类型。但是 boolean 有点特别，虽然 Java 虚拟机也把 boolean 看作是基本类型，但是指令集对 boolean 只提供有限的支持。当编译器把 Java 源码编译为字节码时，它会用 int 或 byte 来表示 boolean。在 Java 虚拟机中，false 是由整数 0 来表示的，所有非 0 整数都表示 true，设计 boolean 值的操作

则会使用 int。另外，boolean 数组是当作 byte 数组来访问的，但是在“堆”区，它也可以被标识为位域。

除了 boolean 类型以外，Java 语言中的基本类型构成了 Java 虚拟机中的数值类型。虚拟机中的数值类型分为两种：整数类型(包括 byte、short、int、long、char)和浮点数类型(包括 float 和 double)和 Java 语言一样，Java 虚拟机的基本类型的值域在任何地方都是一致的，比如，不管底层的主机平台是什么，一个 long 在任何虚拟机中总是一个 64 位二进制补码表示的有符号整数。

Java 虚拟机中还有一个只在内部使用的基本类型：returnAdress，Java 程序员不能使用这个类型，这个基本类型被用来实现 Java 程序中的 finally 子句。

Java 虚拟机的引用类型被统称为“引用”(reference)，主要有如下三种引用类型。

- 类类型：类类型的值是对类实例的引用。
- 接口类型：接口类型的值，则是对实现了该接口的某个类实例的引用。
- 数组类型：数组类型的值是对数组对象的引用，在 Java 虚拟机中，数组是一个真正的对象。

上述三种类型的值都是对动态创建对象的引用。除此之外，Java 虚拟机还有一种特殊的引用值 null，它表示该引用变量没有引用任何对象。

Java 虚拟机规范定义了每一种数据类型的取值范围，但是却没有定义它们的位宽。存储这些类型的值所需的占位宽度，是由具体的虚拟机实现的设计者决定的。关于 Java 虚拟机数据类型的取值范围，具体说明如下。

1. 整形类型和整型值的取值范围

整形类型和整型值的取值范围如下：

- byte 类型：取值范围是从-128～127(-27～27-1)，包括-128 和 127。
- short 类型：取值范围是-32768～32767(-215～215-1)，包括-32768 和 32767。
- int 类型：取值范围是-2147483648～2147483647(-231～231-1)，包括-2147483648 和 2147483647。
- long 类型：取值范围是-9223372036854775808～9223372036854775807(-263 至 263-1)，包括-9223372036854775808 和 9223372036854775807。
- char 类型：取值范围是 0～65535，包括 0 和 65535。

2. 浮点类型、取值集合和浮点值

浮点类型包含 32 位单精度的 float 类型和 64 位双精度的 double 类型两种，浮点数除了包括正负带符号可数的数值，还包括了正负零、正负无穷大和一个特殊的“非数字”标识(Not-a-Number，下文用 NaN 表示)。NaN 值用于表示某些无效的运算操作，例如除数为零等情况。

所有 Java 虚拟机的实现都必须支持两种标准的浮点数值集合：单精度浮点数集合和双精度浮点数集合。

3．returnAddress 类型和值

returnAddress 类型会被 Java 虚拟机的 jsr、ret 和 jsr_w 指令所使用。returnAddress 类型的值指向一条虚拟机指令的操作码。与前面介绍的那些数值类的原始类型不同，returnAddress 类型在 Java 语言之中并不存在相应的类型，也无法在程序运行期间更改 returnAddress 类型的值。

4．boolean 类型

Java 虚拟机不提供操作 boolean 类型的字节码指令，程序在编译后 boolean 类型都转化成了 int 操作。但是 Java 虚拟机支持 boolean 类型的数组的访问和修改，共用 byte 类型数组的字节码指令。

5.3.2 “字”

在 Java 虚拟机中，最基本的数据单元就是字(Word)，它的大小是由每个虚拟机实现的设计者来决定的。字长必须足够大，至少是一个字单元就足以持有 byte、short、int、char、float、returnAddress 或者 reference 类型的值，而两个字单元就足以持有 long 或者 double 类型的值。因此，虚拟机实现的设计者至少得选择 32 位作为字长，或者选择更为高效的字长大小。通常根据底层主机平台的指针长度来选择字长。

在 Java 虚拟机规范中，关于运行时数据区的大部分内容，都是基于“字”这个抽象概念的。比如，关于栈帧的两个部分——局部变量和操作数栈——都是按照“字”来定义的。这些内存区与能够容纳任何虚拟机数据类型的值，当把这些值放到局部变量或者操作数栈中时，它将占用一个或两个字单元。

在运行时，Java 程序无法侦测到底层虚拟机的字长大小。同样虚拟机的字长大小也不会影响程序的行为——它仅仅是虚拟机实现的内部属性。

5.3.3 类装载器子系统

在 Java 虚拟机中，负责查找并装载类型的那部分被称为类装载器子系统。Java 虚拟机有两种类装载器：启动类装载器和用户自定义类装载器。其中前者是 Java 虚拟机实现的一部分，后者则是 Java 程序的一部分。由不同的类装载器装载的类将被放在虚拟机内部的不同命名空间中。

类装载器子系统设计 Java 虚拟机的其他几个组成部分，以及几个来自 Java。Lang 库的类。比如用户自定义的类装载器是普通的 Java 对象，它的类必须派生自 java.lang.ClassLoader 类。ClassLoader 中定义的方法为程序提供了访问类装载器机制的接口。此外，对于每一个被装载的类型，Java 虚拟机都会为它创建一个 java.Lang.Class 类的实力来代表该类型。和所有其他对象一样，用户自定义类装载器以及 Class 类的实例都放在内存区中的堆中，而装载的类型信息则都位于方法区。

装载、连接和初始化类装载器子系统除了要定位和导入二进制 class 文件外，还必须负责验证被导入类的正确性，为类变量分配并初始化内存，以及帮助解析符号引用。这些动作必须严格按照以下顺序进行：

(1) 装载：查找并装载类型的二进制数据。

(2) 连接：执行验证、准备，以及解析(可选)。

- 验证：确保被导入类型的正确性。
- 准备：为类变量分配内存，并将其初始化为默认值。
- 解析：把类型中的符号引用转换为直接引用。

(3) 初始化：把类变量初始化为正确初始值。

(4) 启动类装载器。

只要是符合 Java class 文件格式的二进制文件，Java 虚拟机实现都必须能够从中辨别并装载其中的类和接口。某些虚拟机实现也可以识别其他非规范的二进制格式文件，但它必须能够辨别 class 文件。

每个 Java 虚拟机实现都必须有一个启动类装载器，它知道怎么装载受信任的类，比如 JavaAPI 的 class 文件。Java 虚拟机规范并未规定启动类装载器如何去寻找 class 文件，这又是一件保留给具体的实现设计者去决定的事情。

只要给定某个类型的全限定名，启动类装载器就必须能够以某种方式得到定义该类型的数据。例如 JDK 1.1 会首先逐个搜索用户在 CLASSPATH 环境变量中定义的目录，直到找到一个名为“该类型名+.class”的文件为止。除非该类型属于某个未命名的包，否则启动类装载器会在 CLASSPATH 包含的目录下的子目录中寻找这样一个文件，这些子目录的路径名是根据类型的包名称构建的。比如，如果启动类装载器正在搜索这样一个类 java.lang.Object，那么它将在每一个 CLASSPATH 包含的目录下，查找类似 java\lang 的一个子目录，以及其中的 Object.class 文件。

而在 JDK 的后期版本中，启动类装载器将只在系统类的安装路径中查找要装入的类：而搜索 CLASSPATH 的任务，现在交给了系统类装载器——它是一个自定义的类装载器，当虚拟机启动时就被自动创建了。

(5) 用户自定义类装载器。

尽管“用户自定义类装载器”本身是 Java 程序的一部分，但类 ClassLoader 中的 4 个方法是通往 Java 虚拟机的通道：

```
protected final Class<?> defineClass(String name, byte[] b, int off, int
len);
protected final Class<?> defineClass(String name, byte[] b, int off, int
len, ProtectionDomain protectionDomain);
protected final Class<?> findSystemClass(String name);
protected final void resolveClass(Class<?> c);
```

任何 Java 虚拟机实现都必须把这些方法连接到内部的类装载器子系统中。

两个被重载的 defineCalss()方法都要接受一个名为 data[]的字节数组作为输入参数，并且在 data[offset]到 data[offset+length]之间的二进制数据必须符合 Javaclass 文件格式——它表示一个新的可用类型。而 name 参数是个字符串，它支持该类型的全限定名。当使用第一个 defineClass()时，该类型将被赋予默认的保护域。使用第二个 definClass()时，该类型的保护域将由它的 protectionDomain 参数指定。每个 Java 虚拟机实现都必须保证 ClassLoader 类的 defineClass()方法能够把新类型导入到方法区中。

方法 findSystemClass()接受一个字符串作为参数，它支持将被装入类型的全限定名。在 1.0 版本和 1.1 版本中，此方法会通过启动类装载器类装载指定类型。如果启动类装载器装载完成，它会返回对 Class 对象(该对象描述了该类型)的引用。如果没有找到相应的 class 文件，它会抛出 ClassNotFoundException 异常。而在后面的版本中，方法 findSystemClass()使用系统类装载器来装载指定类型。任何 Java 虚拟机实现都必须保证方法 findSystemClass()能够以这种方式调用启动类装载器或者系统类装载器。

方法 resolverClass()接受一个 Class 实例的引用作为参数，它将对该 Class 实例表示的类型执行连接动作。而方法 defineClass()则只负责装载。当方法 defineClass()返回一个 Class 实例时，也就表示指定的 class 文件已经被找到并装载到方法区了，但是却不一定被连接和初始化。Java 虚拟机实现必须保证 CLassLoader 类的 resolveClass()方法能够让类装载器子系统执行连接动作。

(6) 命名空间。

每个类装载器都有自己的命名空间，其中维护着由它装载到额的类型。所以一个 Java 程序可以多次装载具有同一个全限定名的多个类型。这样一个类型的全限定名就不足以确定在一个 Java 虚拟机中的唯一性。因此，当多个类装载器装载了同名的类型时，为了唯一地标识该类型，还要在类型名称前加上装载该类型的类装载器的标识。

Java 虚拟机中的命名空间，其实是解析过程的结果。对于每一个被装载的类型，Java 虚拟机都会记录装载它的类装载器。当虚拟机解析一个类到另一个类的符号引用时，它需要被引用类的类装载器。

5.3.4　方法区

在 Java 虚拟机中，关于被装载类型的信息存储在一个逻辑上被称为方法区的内存中，当虚拟机装载某个类型时，它使用类装载器定位相应的 class 文件，然后读入这个 class 文件(一个线性二进制数据流)，然后将它传输到虚拟机中。紧接着虚拟机提取其中的类型信息，并将这些信息存储到方法区。该类型中的类变量同样也存储在方法区中。

Java 虚拟机在内部如何存储类型信息，这是由具体实现的设计者来决定的。例如在 class 文件中，多字节值总是以高位在前的顺序存储。但是当这些数据引入到方法区后，虚拟机可以以任何方式存储它。假设某个实现是运行在低位优先的处理器上，那么它很可能会把多字节值以低位优先的顺序存储到方法区中。

当虚拟机运行 Java 程序时，它会查找使用存储在方法区中的类型信息。设计者应当为类型信息的内部表示设计恰当的数据结构，以尽可能在保持虚拟机小巧紧凑的同时加快程序的运行效率。如果正在设计一个需要在少量内存的限制中操作的实现，设计者可能会决定以牺牲某些运行速度来换取紧凑性。另外一个方面，如果设计一个将在虚拟内存系统中运行的实现，设计者可能会决定在方法区中保存一些冗余信息，以加快执行速度。如果底层主机没有提供虚拟内存，但是提供了一个硬盘，设计者可能会在实现中创建一个虚拟内存系统。此时 Java 程序员可以根据目标平台的资源限制和需求，在空间和时间上做出权衡，选择实现什么样的数据结果和数据组织。

由于所有线程都共享方法区，因此它们对方法区数据的访问必须被设计为是线程安全

的。比如，假设同时有两个线程都企图访问一个名为 EXAM 的类，而这个类还没有被装入虚拟机，那么，这时只应该有一个线程去装载它，而另一个线程则只能等待。

方法区的大小不必是固定的，虚拟机可以根据应用的需要动态调整。同样，方法区也不必是连续的，方法区可以在一个堆中自由分配。另外，虚拟机也可以允许用户或者程序员指定方法区的初始大小以及最小和最大尺寸等。

方法区也可以被垃圾收集，因为虚拟机允许通过用户定义的类装载器来动态扩展 Java 程序，因此一些类也会成为程序“不再引用”的类。当某个类变为不再被引用的类时，Java 虚拟机可以卸载这个类(垃圾收集)，从而使方法区占据的内存保持最小。类的卸载以及一个类变为“不再被引用”的必需条件。

- 类型信息：对每个装载的类型，虚拟机都会在方法区中存储以下类型信息：
- 这个类型的全限定名。
- 这个类型的直接超类的全限定名(除非这个类是 java.lang.Object，它没有超类)。
- 这个类型是类类型还是接口类型。
- 这个类型的访问修饰符(public、abstract 或 final 的某个子集)。
- 任何直接超接口的全限定名的有序列表。

在 Java class 文件和虚拟机中，类型名总是以全限定名出现。在 Java 源代码中，全限定名由类所属包的名称加一个“.”，再加上类名组成。例如，类 Object 的所属包为 java.lang，那么它的全限定名应该是 java.lang.Object，但在 class 文件里，所有的“.”都被斜杠“/”代替，这样就成为 java/lang/Object。至于全限定名在方法区中的表示，则因不同的设计者有不同的选择而不同，可以用任何形式和数据结构来表示。

除了上面列出的基本类型信息外，虚拟机还得为每个被装载的类型存储以下信息：

- 该类型的常量池。
- 字段信息。
- 方法信息。
- 除了常量意外的所有类(静态)变量。
- 一个到类 CLassLoader 的引用。
- 一个到 Class 类的引用。

在接下来的内容中，将简要介绍上述数据类型。

1) 常量池

虚拟机必须为每个被装载的类型维护一个常量池。常量池就是该类型所用常量的一个有序集合，包括直接常量(Stringinteger 和 floatingpoint 常量)和对其他类型、字段和方法的符号引用。池中的数据项就像数组一样是通过索引访问的。因为常量池存储了相应类型所用到的所有类型、字段和方法的符号引用，所以它在 Java 程序的动态连接中起着核心作用。

字段信息：对于类型中生命的每一个字段，方法区中必须保存下面的信息。除此之外，这些字段在类或者接口中的生命顺序也必须保存。下面是字段信息的清单：

- 字段名
- 字段的类型

- 字段的修饰符(public,private,protected,static,final,volatile,transient 的某个子集)

2) 方法信息

对于类型中声明的每一个方法，方法区中必须保存下面的信息。和字段一样，这些方法在类或者接口中的生命顺序也必须保存。下面是方法信息的清单：

- 方法名
- 方法的返回类型(或 void)
- 方法参数的数量和类型(按声明顺序)
- 方法的修饰符

除上面的清单中列出的条目之外，如果某个方法不是抽象的和本地的，它还必须保存下列信息：

- 方法的字节码(bytecodes)
- 操作数栈和该方法的栈帧中的局部变量区的大小
- 异常表

类(静态)变量是由所有类实例共享的，但是即使没有任何类实例，它也可以被访问。这些变量只与类有关——而非类的实例，因此它们总是作为类型信息的一部分而存储在方法区。除了在类中生命的编译时常量外，虚拟机在使用某个类之前，必须在方法区中为这些类分配空间。

而编译时常量(就是那些用 final 声明以及用编译时已知的值初始化的类变量)则和一般的类变量的处理方式不同，每个使用编译时常量的类型都会复制它的所有常量到自己的常量池中，或嵌入到它的字节码流中。作为常量池或字节码流的一部分，编译时常量保存在方法区中——就和一般的类变量一样。但是当一般的类变量作为声明它们的类型的一部分数据而保存的时候，编译时常量作为使用它们的类型的一部分而保存。

指向 ClassLoader 类的引用每个类型被装载的时候，虚拟机必须跟踪它是由启动类装载器还是由用户自定义类装载器装载的。如果是用户自定义类装载器装载的，那么虚拟机必须在类型信息中存储对该类装载器的引用。这是作为方法表中的类型数据的一部分保存的。

虚拟机会在动态连接期间使用这个信息。当某个类型引用另一个类型的时候，虚拟机会请求装载发起引用类型的类装载器来装载被引用的类型。这个动态连接的过程，对于虚拟机分离命名空间的方式也是至关重要的。为了能够正确地执行动态连接以及维护多个命名空间，虚拟机需要在方法表中得知每个类都是由哪个类装载器装载的。

3) 指向 Class 类的引用

对于每一个被装载的类型(不管是类还是接口)，虚拟机都会相应地为它创建一个 java.lang.Class 类的实例，而且虚拟机还必须以某种方式把这个实例和存储在方法区中的类型数据关联起来。

在你的 Java 程序中，你可以得到并使用指向 Class 对象的引用。Class 类中的一个静态方法可以让用户得到任何已装载的类的 Class 实例的引用。

```
public static Class forName(String className);
```

比如，如果调用 forName(“java.lang.Object”)，那么将得到一个代表 java.lang.Object 的 Class 对象的引用。可以使用 forName()来得到代表任何包中任何类型的 Class 对象的引用，只要这个类型可以被(或者已经被)装载到当前命名空间中。如果虚拟机无法把请求的类型装载到当前命名空间，那么 forName()会抛出 ClassNotFoundException 异常。

另一个得到 Class 对象引用的方法是，可以调用任何对象引用的 getClass()方法。这个方法被来自 Object 类本身的所有对象继承：

```
Public final Class getClass();
```

比如，如果你有一个到 java.lang.integer 类的对象的引用，那么你只需要简单地调用 Integer 对象引用的 getClass()方法，就可以得到表示 java.lang.Integer 类的 Class 对象。

给出一个指向 Class 对象的引用，就可以通过 Class 类中定义的方法来找出这个类型的相关信息。如果查看这些方法，会很快意识到，Class 类使得运行程序可以访问方法区中保存的信息。下面是在 Class 类中声明的方法：

```
Public String getName();
Public Class getSupperClass();
Public Boolean isInterface();
Public Class[] getInterfaces();
Public ClassLoader getClassLoader();
```

这些方法仅能够返回已装载类型的信息。getName()返回类型的全限定名，getSupperClass()返回类型的直接超类的 Class 实例。如果类型是 java.lang.Object 类或者是一个接口，它们都没有超类，getSupperClass()返回 null。isInterface()判断该类型是否是接口，如果 Class 对象描述一个接口就返回 true；如果它描述一个类则返回 false，getInterfaces()返回一个 Class 对象数组，其中每个 Class 对象对应一个直接超接口，超接口在数组中以类型声明超接口的顺序出现。如果该类型没有直接超接口，getInterfaces()则返回一个长度为零的数组。getClassLoader()返回装载该类型的 ClassLoader 对象的引用，如果类型是由启动类装载器装载的，则返回 null。所有这些信息都直接从方法区中获得。

方法表为了尽可能提高访问效率，设计者必须仔细设计存储在方法区中的类型信息的数据结构，因此，除了以上讨论的原始类型信息，实现中还可能包括其他数据结构以加快访问原始数据的速度，比如方法表。虚拟机对每个装载的非抽象类，都声称了一个方法表，把它作为类信息的一部分保存在方法区。方法表是一个数组，它的元素是所有它的实例可能被调用的实例方法的直接引用，包括那些从超类继承过来的实例方法。

4) 方法区使用示例

接下来将通过一个简单示例来展示虚拟机如何使用方法区中的信息，看如下类的实现代码：

```
public class EXAM {
     private int speed = 5;// 5kilometers per hour
     void flow() {
     }
}
class Mmm {
```

```
    public static void main(String[]args) {
        EXAMEXAM = new EXAM();
        EXAM.flow();
    }
}
```

下面的段落描述了某个实现中是如何执行 Mmm 程序中 main()方法的字节码中第一条指令的。不同的虚拟机实现可能会用完全不同的方法来操作，下面描述的可能只是其中的一种——但并不是仅有的一种，接下来看 Java 虚拟机是如何执行 Mmm 程序中 main()方法的第一条指令。

要运行 Mmm 程序，首先需要以某种“依赖于实现的”方式告诉虚拟机“Mmm”这个名字。之后，虚拟机将找到并读入相应的 class 文件“Mmm.class”，然后它会从导入的 class 文件里的二进制数据中提取类型信息并放到方法区中。通过执行保存在方法区中的字节码，虚拟机开始执行 main()方法，在执行时，它会一直持有指向当前类的常量池的指针。

5.3.5　堆

Java 程序在运行时创建的所有类实例或数组都放在同一个堆中。而一个 Java 虚拟机实例中只存在一个堆空间，因此所有线程都将共享这个堆。又由于一个 Java 程序独占一个 Java 虚拟机实例，因而每个 Java 程序都有它自己的堆空间——它们不会彼此干扰。但是同一个 Java 程序的多个线程却共享着同一个堆空间，在这种情况下，就得考虑多线程访问对象(堆数据)的同步问题了。

Java 虚拟机有一条在堆中分配新对象的指令，却没有释放内存的指令。正如我们无法使用 Java 代码去明确释放一个对象一样，字节码指令也没有对应的功能。虚拟机自己负责决定如何以及何时释放不再被运行的程序引用的对象所占据的内存。程序本身不用去考虑何时需回收对象所占用的内存，通常虚拟机把这个人物交给垃圾收集器。

1) 垃圾收集

垃圾收集器的主要工作就是自动回收不再被运行的程序引用的对象所占用的内存。此外，它也可能去移动那些还在使用的对象，以此减少堆碎片。

Java 虚拟机规范并没有强制规定垃圾收集器，它只要求虚拟机实现必须“以某种方式”管理自己的堆空间。举个例子，某个实现可能只有固定大小的堆空间可用，当空间填满，它就简单地抛出 OutOfMemory 异常，根本不去考虑回收垃圾对象的问题。这样的一个实现虽然简陋，但却是符合规范的。总之，Java 虚拟机规范并没有规定具体的实现必须为 Java 程序准备多少内存，也没有说它必须怎么管理自己的堆空间，它仅仅告诉实现的设计者：Java 程序需要从堆中为对象肥培空间，并且程序本身不会主动释放它。因此堆空间的管理(包括垃圾收集)问题得由设计者自行去考虑处理方式。

Java 虚拟机规范没有指定垃圾收集应该采用什么技术。这些都由虚拟机的设计者根据他们的目标、考虑所受的限制、用自己的能力去决定什么才是最好的技术。因为到对象的引用可能很多地方都存在，如 java 栈、堆、方法区、本地方法栈，所以垃圾收集技术的使用在很大程度上会影响到运行时数据区的设计。

和方法区一样，堆空间也不必是连续的内存区。在程序运行时，它可以动态扩展或收缩。事实上，一个实现的方法区可以在堆顶实现。换句话说，就是当虚拟机需要为一个新装载的类分配内存时，类型信息和实际对象可以都在同一个堆上。因此，负责回收无用对象的垃圾收集器可能也要负责无用类的释放(卸载)。另外，某些实现可能也允许用户或程序员指定堆的初始大小、最大最小值等。

2) 对象的内部表示

Java 虚拟机规范并没有规定 Java 对象在堆中是如何表示的。对象的内部表示也影响着整个堆以及垃圾收集器的设计，它由虚拟机的实现者决定。

Java 对象中包含的基本数据由它所属的类及其所有超类声明的实例变量组成。只要有一个对象引用，虚拟机就必须能够快速地定位对象实例的数据。另外，它也必须能通过该对象引用访问相应的类数据(存储于方法区的类型信息)。因此在对象中通常会有一个指向方法区的指针。

一种可能的堆空间设计就是，把堆分为两部分：一个句柄池和一个对象池。而一个对象引用就是一个指向句柄池的本地指针。句柄池的每个条目有两个部分：一个指向对象实例变量的指针，一个指向方法区类型数据的指针。这种设计的好处是有利于堆碎片的整理，当移动对象池中的对象时，句柄部分只需要更改一下指针指向对象的新地址就可以了——就是在句柄池中的那个指针；缺点是每次访问对象的实例变量都要经过两次指针传递。这种对象表示的方法如图 5-7 所示。

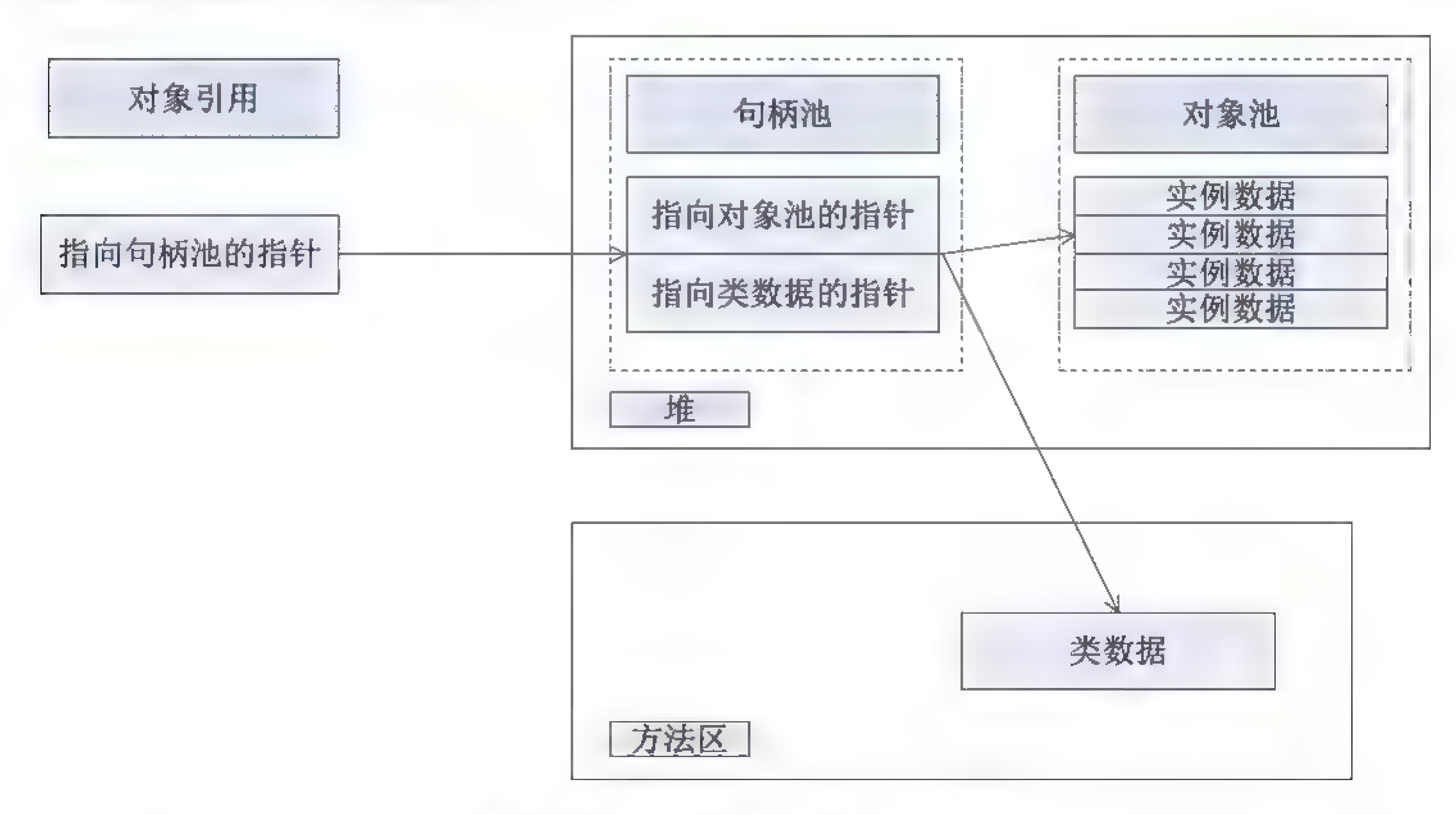


图 5-7　划分为对象池和方法池的对象

另一种设计方法是使对象指针直接指向一组数据，而该数据包括对象实例数据以及指向方法区中类数据的指针，如图 5-8 所示。这样设计的优缺点正好与前面的方法相反，它只需要一个指针就可以访问对象的实例数据，但是移动对象就变得更加复杂。当使用这种

堆的虚拟机为了减少内存碎片而移动对象的时候，它必须在整个运行时数据区中更新指向被移动对象的引用。

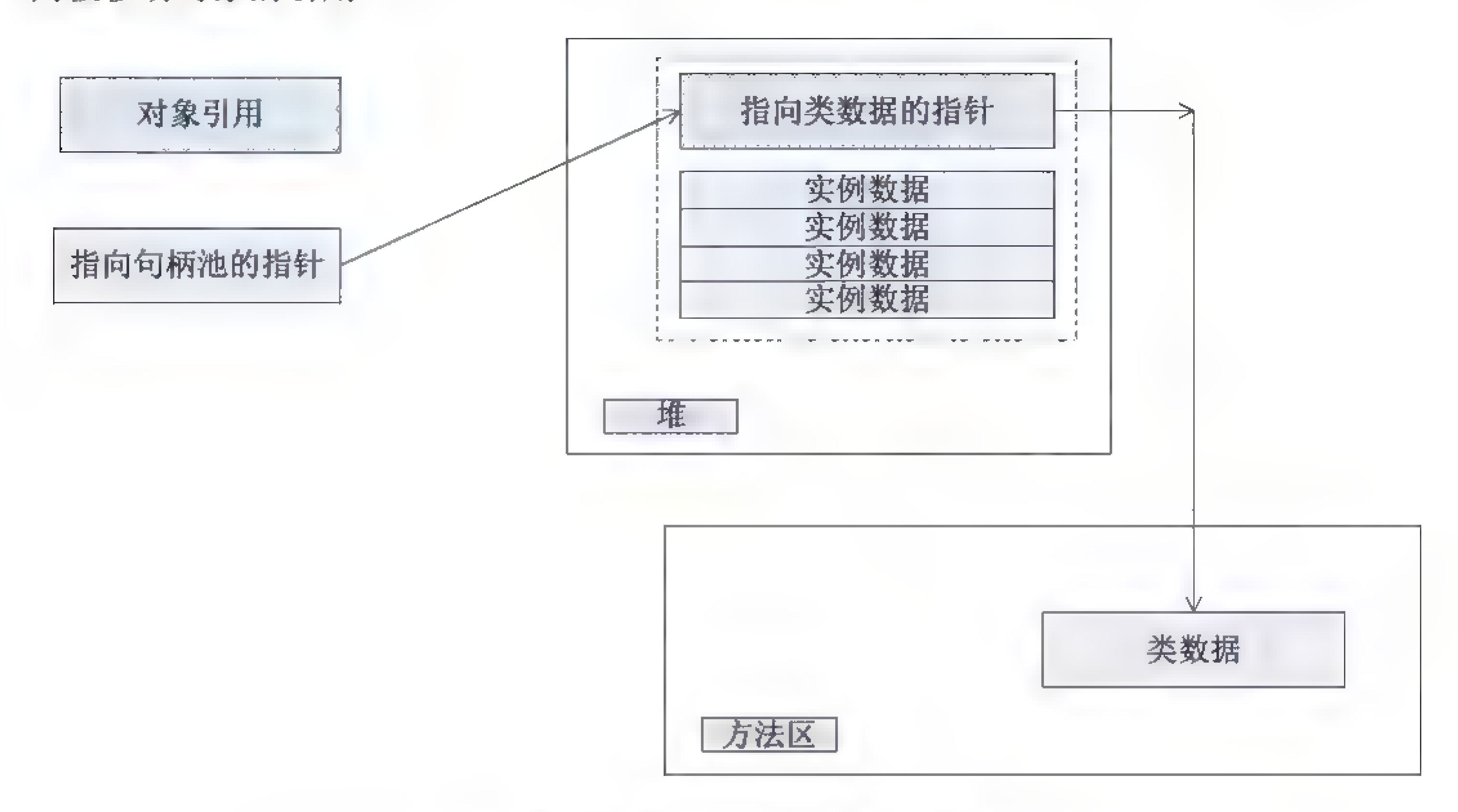


图 5-8　保持对象和数据在一起

有如下几个理由要求虚拟机必须能够通过对象引用得到类(类型)数据：当程序运行时需要转换某个对象引用为另一种类型时，虚拟机必须要检查这种转换是否被允许，被转换的对象是否的确是被引用的对象或者它的超类型。当程序在执行 instanceof 操作时，虚拟机也进行了同样的检查。在这两种情况下，虚拟机都需要查看被引用的对象的类数据。最后，当程序中调用某个实例方法时，虚拟机必须进行动态绑定，换句话说，它不能按照引用的类型来决定将要调用的方法，而必须根据对象的实际类。为此，虚拟机必须再次通过对象的引用去访问类数据。

不管虚拟机的实现使用什么样的对象表示法，很可能每个对象都有一个方法表，因为方法表加快了调用实例方法时的效率，从而对 java 虚拟机实现的整体性能起着非常重要的正面作用。但是 Java 虚拟机规范并未要求必须使用方法表，所以并不是所有实现中都会使用它。比如那些具有严格内存资源限制的实现，或许他们根本不可能有足够的额外内存资源来存储方法表。如果一个实现使用方法表，那么仅仅使用一个指向对象的引用，就可以很快地访问对象的方法表。

图 5-9 展示了一种把方法表和对象引用联系起来的实现方式：每个对象的数据都包含一个指向特殊数据结构的指针，这个数据结构位于方法区，它包括两部分：

- 一个指向方法区对应类数据的指针。
- 此对象的方法表。

方法表是一个指针数组，其中的每一项都是一个指向“实例方法数据”的指针，实例方法可以被那类的对象调用，方法表指向的实例方法数据包括以下信息：

- 此方法的操作数栈和局部变量去的大小。
- 此方法的字节码。
- 异常表。

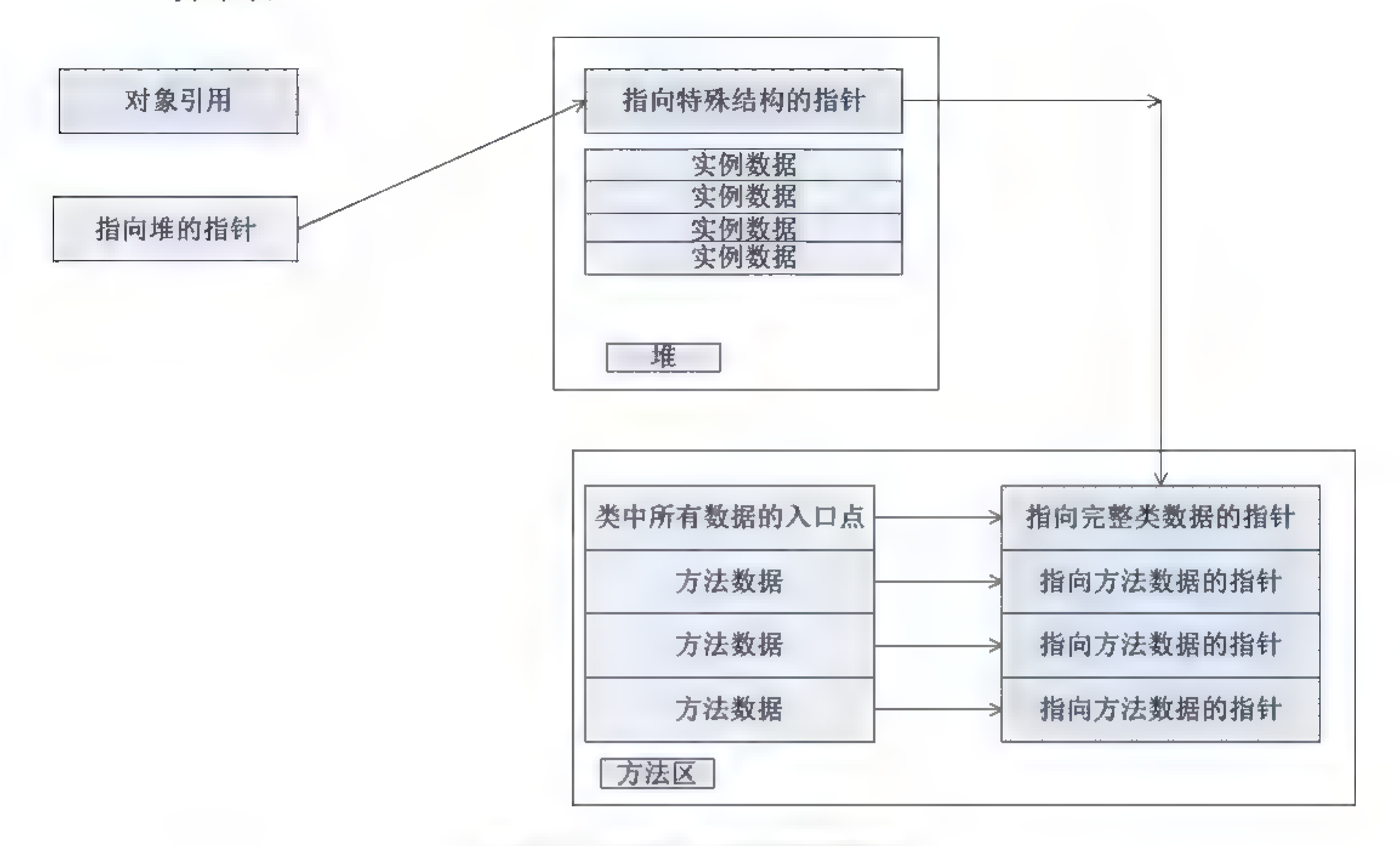


图 5-9　保持方法区数据随手可用

这些信息足够虚拟机去调用一个方法了，方法表中包含有方法指针——指向类或其超类声明的方法的数据，也就是说，方法表所指向的方法可能是此类声明的，也可能是它继承下来的。

如果读者熟悉 C++的内部机制，就会发现这和 C++的 VTBL(C++对象的虚拟表)非常相似。在 C++中，对象由实例数据和一组指向对象可以调用的虚拟函数的指针组成。Java 虚拟机也可以采用这种方法，虚拟机可以在堆中为每个对象都附加一个方法表，这样较之图 5-9 的方法会占用更多的内存，但可轻微提高一些效率。一般来说，改方案只是用于内存足够充裕的系统。

图 5-7 和图 5-8 中显示的还有另一种数据，堆上的对象数据中还有一个逻辑部分，那就是对象锁，这是一个互斥对象。虚拟机中的每个对象都有一个对象锁，它被用于协调多个线程访问同一个对象时的同步。在任何时刻，只能有一个线程“拥有”这个对象锁，因此只有这个县策划那个才能访问该对象的数据。此时其他希望访问这个对象的线程只能等待，直到拥有对象锁的线程释放锁，当某个线程拥有一个对象锁后，可以继续对这个锁追加请求。但请求几次，必须对应地释放几次，之后才能轮到其他线程。比如一个线程请求了三次锁，在它释放三次锁之前，它一直保持“拥有”这个锁。

很多对象在其整个生命周期内都没有被任何线程加锁。在线程实际请求某个对象的锁之前，实现对象锁需要的数据是不必要的。这样正如图 5-7 和图 5-8 所示，很多实现不在对象自身内部保存一个指向锁数据的指针。而只有当第一次需要加锁的时候才分配对应的

锁数据，但这时虚拟机需要用某种间接方法来联系对象数据和对应的锁数据，例如把锁数据放在一个以对象地址为索引的搜索树中。

除了实现锁锁需要的数据外，每个 Java 对象逻辑上还与实现等待集合(Wait Set)的数据相关联。锁是用来实现多个线程对共享数据的互斥访问的，而等待集合是用来让多个线程为完成一个共同目标而协调工作的。

等待集合由等待方法和通知方法联合使用。每个类都从 Object 那里继承了三个等待方法(三个名为 wait()的重载方法)和两个通知方法(notify()和 notifyAll())。当某个线程在一个对象上调用等待方法时，虚拟机就阻塞这个线程，并把它放在了这个对象的等待集合中。知道另一个线程在同一个对象上调用通知方法，虚拟机才会在之后的某个时刻唤醒一个或多个正在等待集合中被阻塞的线程。正像锁数据一样，在实际调用对象的等待方法或通知方法之前，实现对象的等待集合的数据并不是必须的。因此许多虚拟机实现都把等待集合数据与实际对象数据分开，只有在需要时才为此对象创建同步数据

最后一种数据类型——可以作为堆中某个对象映像的一部分，是与垃圾收集器有关的数据。垃圾收集器必须(以某种方式)跟踪程序引用的每个对象，这个人物不可避免地要附加一些数据给这些对象，数据的类型要视垃圾收集使用的算法而定。例如，加入垃圾收集器使用“标记并清除”算法，这就需要能够标记对象能否被引用。此外，对于不再被引用的对象，还需要指明它的终结算法(Finalizer)是否已经运行过了。像线程锁一样，这些数据也可以放在对象数据外。有一些垃圾收集技术只在垃圾收集器运行时需要额外数据。例如“标记并清除”算法就使用一个独立的位图来标记对象的引用情况。

除了标记对象的引用情况外，垃圾收集器还要区分对象是否调用了终结方法。对于在其类中声明了终结方法的对象，在回收它之前，垃圾收集器必须调用它的终结方法。Java 语言规范指出，垃圾收集器对每个对象只能调用一次终结方法，但是允许终结方法复活这个对象，即允许对象再次被引用。这样当这个对象再次被回收时，就不再调用终结方法了。需要终结方法的对象不多，而需要复活的更少，所以对一个对象回收两次的情况很少见。这种用来标志终结方法的数据虽然在逻辑上是对象的一部分，但通常实现上不随对象保存在堆中。大部分情况下，垃圾收集器会在一个单独的空间保存这个信息。

数组的内部表示：在 Java 中，数组是真正的对象。和其他对象一样，数组总是存储在堆中。同样，和普通对象一样，实现的设计者将决定数组在堆中的表示形式。

和其他所有对象一样，数组也拥有一个与它们的类相关联的 Class 实例，所有具有相同维度和类型的数组都是同一个类的实例，而不管数组的长度是多少。例如，一个包含 3 个 int 整数的数组和一个包含 300 个 int 整数的数组拥有同一个类。数组的长度只与实例数据有关。

数组类的名称由两部分组成：每一维用一个方括号表示。多为数组被标识为数组的数组。比如，int 类型的二维数组，将表示为一个一维数组，其中的每个元素是一个一维数组的引用。

在堆中的每个数组对象还必须保存的数据是数组的长度、数组数据，以及某些指向数组的类数据的引用。虚拟机必须能够通过一个数组对象的引用得到此数组的长度，通过索引访问其元素，调用所有数组的直接超类 Object 声明的方法等。

5.3.6 程序计数器

对于一个运行中的Java程序来说，其中的每一个线程都有自己的PC(程序计数器)寄存器，它是在该线程启动时创建的。PC 寄存器的大小是一个字长，因此它既能持有一个本地指针，也能够持有一个 returnAddress。当线程执行某个 Java 方法时，PC 寄存器的内容总是笑一天将被执行指令的“地址”，这里的“地址”可以是一个本地指针，也可以是在方法字节码中相对于该方法起始指令的偏移量。如果该线程正在执行一个本地方法，那么此时 PC 寄存器的值是“undefined”。

5.3.7 Java 栈

每当启动一个新线程时，Java 虚拟机都会为它分配一个 Java 栈。前面我们曾经提到，Java 栈以帧为单位保存线程的运行状态。虚拟机只会直接对 Java 栈执行两种操作：以帧为单位的压栈和出栈。

某个线程正在执行的方法被称为该线程的当前方法，当前方法使用的栈帧成为当前帧，当前方法所属的类成为当前类，当前类的常量池成为当前常量池。在线程执行一个方法时，它会跟踪当前类和当前常量池。此外，当虚拟机遇到栈内操作指令时，它对当前帧内数据执行操作。

每当线程调用一个 Java 方法时，虚拟机都会在该线程的 Java 栈中压入一个新帧。而这个新帧自然就成为了当前帧。在执行这个方法时，它使用这个帧来存储参数、局部变量、中间运算结果等等数据。

Java 方法可以以两种方式完成：一种是通过 return 返回的，称为正常返回；一种是通过抛出异常而异常中止的。不管以哪种方式返回，虚拟机都会将当前帧弹出 Java 栈，然后释放掉，这样上一个方法的帧就成为当前帧了。

Java 栈上的所有数据都是此线程私有的。任何线程都不能访问另一个线程的栈数据，因此我们不需要考虑多线程情况下栈数据的访问同步问题。当一个线程调用一个方法时，方法的局部变量保存在调用线程 Java 栈的帧中。只有一个线程总是访问哪些局部变量，即调用方法的线程。

像方法区和堆一样，Java 栈和帧在内存中也不必是连续的。帧可以分布在连续的栈里，也可以分布在堆里，或者二者兼而有之。表示 Java 栈和栈帧的实际数据结构由虚拟机的实现者决定。某些实现允许用户指定 Java 栈的初始大小和最大最小值。

5.3.8 栈帧

栈帧由局部变量区、操作数栈和帧数据区构成。局部变量区和操作数栈的大小要视对应的方法而定，它们是按字长计算的。编译器在编译时就确定了这些值并放在 class 文件中。而帧数据区的大小依赖于具体的实现。

当虚拟机调用一个 Java 方法时，它从对应类的类型信息中得到此方法的局部变量区和操作数栈的大小，并据此分配栈帧内存，然后压入 Java 栈中。

1) 局部变量区

Java 栈帧的局部变量区被组织为一个以字长为单位，从 0 开始计数的数组。字节码指令通过从 0 开始的索引来使用其中的数据。类型为 int、float、reference 和 returnAddress 的值在数组中只占据一项，而类型为 byte、short 和 char 的值在存入数组前都被转换为 int 值，因而同样占据一项。但是类型为 long 和 double 的值在数组中却占连续的两项。

在访问局部变量中的 long 和 double 值的时候，指令只需指出连续两项中第一项的索引值。例如某个 long 值占据第三四项，那么指令会取索引为 3 的 long 值。局部变量区的所有值都是字对齐的，long 和 double 这样占据两项数组元素的值同样可以起始于任何索引。

局部变量区包含对应方法的参数和局部变量。编译器首先按声明的顺序把这些参数放入局部变量数组。例如下面的代码描绘了两个方法的局部变量区。

```
public class Example3a {
     public static int runClassMethod(int i, long l, float f, double d,
Object o, byte b) {
          return 0;
     }
     public intrunMethod(char c, double d, short s, boolean b) {
          return 0;
     }
}
```

在上述方法 runMethod()中，局部变量中第一个参数是一个 reference(引用)类型，尽管在方法源代码中没有显式声明这个参数，但这个参数 this 对于任何一个实例方法都是隐含加入的，它用来表示调用该方法的对象本身，与此相反，方法 runClassMethod()中就没有这个隐含的 this 变量，因为它是一个类方法。类方法只与类相关，而与具体的对象无关，不能直接通过类方法访问类实例的变量，因为在方法调用的时候没有关联到一个具体实例。

我们注意到，在源代码中的 byte、short、char 和 boolean 在局部变量区都被转换成了 int，在操作数栈中也一样，前面我们曾经说过，虚拟机并不直接支持 boolean 类型，因此 Java 编译器总是用 int 来表示 boolean，但 Java 虚拟机直接支持 byte、short 和 char 类型，这些类型的值既可以作为实例变量或者数组元素存储在局部变量区，也可以作为类变量存储在方法区中。但是在局部变量区和操作数栈中都会被转换成 int 类型的值，它们在栈帧中的时候都是当作 int 来进行处理的。只有当它被存回堆或方法区时，才会转换回原来的类型。

同样需要注意的是作为 runClassMethod()的引用被传递的参数 Objecto。在 Java 中，所有的变量都按引用传递，并且都存储在堆中，永远都不会在局部变量区或操作数栈中发现对象的拷贝，只会有对象引用。

除了 Java 方法的参数(编译器首先严格按照它们的声明顺序放到局部变量数组中，而对于真正的局部变量，它可以任意决定放置顺序，甚至可以用一个索引指代两个局部变量)——比如当两个局部变量的作用域不重叠时，像下面 Example3b 中的局部变量 i 和 j 就是这种情况：在方法的前半段，在 j 开始生效前，0 号索引的入口可以被用来代表 i 在方法的

后半段，i 已经超过了有效作用域，0 号入口就可以用来表示 j 了。

```
class Example3b {
     public void runTwoLoops() {
          for (int i = 0; i <10; i++) {
               System.out.println(i);
          }
          for (int j = 0; j <10; j++) {
               System.out.println(j);
          }
     }
}
```

和其他运行时内存区一样，虚拟机的实现者可以为局部变量区设计任意的数据结构。比如对于怎样把 long 和 double 类型的值存储到两个数组项中，Java 虚拟机规范没有指定。假如某个虚拟机实现的字长为 64 位，这时就可以把整个 long 或 double 数据放在数组中相邻两数组项的低项内，而使高项保持为空。

2) 操作数栈

操作数栈和局部变量区一样，操作数栈也是被组织成一个以字长为单位的数组。但是和前者不同的是，它不是通过索引来访问，而是通过标准的栈操作——压栈和出栈——来访问的。比如，如果某个指令把一个值压入到操作数栈中，稍后另一个指令就可以弹出这个值来使用。

虚拟机在操作数栈中存储数据的方式和在局部变量区中是一样的：如 int、long、float、double、reference 和 returnType 的存储。对于 byte、short 以及 char 类型的值在压入到操作数栈之前，也会被转换为 int。

不同于程序计数器，Java 虚拟机没有寄存器，程序计数器也无法被程序指令直接访问。Java 虚拟机的指令是从操作数栈中而不是从寄存器中取得操作数的，因此它的运行方式是基于栈的而不是基于寄存器的。虽然指令也可以从其他地方取得操作数，比如从字节码流中跟随在操作吗(代表指令的字节)之后的字节中或从常量池中，但是主要还是从操作数栈中获得操作数。

虚拟机把操作数栈作为它的工作区——大多数指令都要从这里弹出数据，执行运算，然后把结果压回操作数栈。比如，iadd 指令就要从操作数栈中弹出两个整数，执行加法运算，其结果又压回到操作数栈中，看看下面的示例，它演示了虚拟机是如何把两个 int 类型的局部变量相加，再把结果保存到第三个局部变量的：

```
Iload 0 //push the int in localvariable 0
Iload 1 //push the int in localvariable 1
Iadd  // pop two ints , add them , push result
Istore_2  //pop int, store into local variable 2
```

在这个字节码序列里，前两个指令 iload 0 和 iload 1 将存储在局部变量中索引为 0 和 1 的整数压入操作数栈中，其后 iadd 指令从操作数栈中弹出那两个整数相加，再将结果压入操作数栈。第四条指令 istore_2 则从操作数栈中弹出结果，并把它存储到局部变量区索引为 2 的位置。

3) 帧数据区

除了局部变量区和操作数栈外，Java 栈帧还需要一些数据来支持常量池解析、正常方法返回以及异常派发机制。这些信息都保存在 Java 栈帧的帧数据区中。

Java 虚拟机中的大多数指令都涉及常量池入口。有些指令仅仅是从常量池中取出数据然后压入 Java 栈(这些数据的类型包括 int、long、float、double 和 String)；还有些指令使用常量池的数据来指示要实例化的类或数组，要访问的字段，或要调用的方法；还有些指令需要常量池中的数据才能确定某个对象是否属于某个类或实现了某个接口。

每当虚拟机要执行某个需要用到常量池数据的指令时，它都会通过帧数据区中指向常量池的指针来访问它。以前讲过，常量池中对类型、字段和方法的引用在开始时都是符号。当虚拟机在常量池中搜索的时候，如果遇到指向类、接口、字段或者方法的入口，假若它们仍然是符号，虚拟机那是才会(也必须)进行解析。

除了用于常量池的解析外，帧数据区还要帮助虚拟机处理 Java 方法的正常结束或异常中止。如果是通过 return 正常结束，虚拟机必须恢复发起调用的方法的栈帧，包括设置 PC 寄存器指向发起调用的方法中的指令——即紧跟着调用了完成方法的指令的下一个指令。假如方法有返回值，虚拟机必须将它压入到发起调用的方法的操作数栈。

为了处理 Java 方法执行期间的异常退出情况，帧数据区还必须保存一个对此方法异常表的引用。异常表定义了在这个方法的字节码中受 catch 子句保护的范围，异常表中的每一项都有一个被 catch 子句保护的代码的起始和结束为止(即 try 子句内部的代码)，可能被 catch 的异常类在常量池中的索引值，以及 catch 子句内的代码开始的位置。

当某个方法抛出异常时，虚拟机根据帧数据区对应的异常表来决定如何处理。如果在异常表中找到了匹配的 catch 子句，就会把控制权转交给 catch 子句内的代码。如果没有发现，方法会立即异常中止。然后虚拟机使用帧数据区的信息回复发起调用的方法的帧，然后在发起调用的方法的上下文中重新抛出同样的异常。

除了上述信息(支持常量池解析、正常方法返回和异常派发的数据)外，虚拟机的实现者也可以将其他信息放入帧数据区，如用于调试的数据等。

4) Java 栈的可能实现方式

程序设计者可以按自己的想法来随意设计 Java 栈，例如一个可能的方式就是从堆中分配每一个帧。例如考虑下面的类：

```
class Example3c {
    public static void addAndPrint() {
        double result = addTwoTypes(1,88.88);
        System.out.println(result);
    }
    public static double addTwoTypes(int i, double d) {
        return i + d;
    }
}
```

在 Java 虚拟机的实现中，每个帧都可以单独从堆中分配。为了调用方法 addTwoTypes()和方法 addAndPrint()。首先把 int 1 和 double 88.88 压入到它的操作数栈中，然后调用 addTwoTypes()方法。

调用 addTwoTypes()的指令指向一项常量池的数据，因此虚拟机在常量池中查找这些数据，这期间如有必要还需要进行解析。

在此需要注意的是，方法 addAndPrint()也要使用常量池才能确定 addTwoTypes()方法，尽管这两个方法是属于一个类的。由此可见，和引用其他类的字段或方法一样，对同一个类的方法和字段的引用初始时也是符号，因此在使用之前同样需要解析。

解析后的常量池数据项将指向方法区中对应方法 addTwoTypes()的信息。虚拟机需要使用这些信息来决定 addTwoTypes()的局部变量区和操作数栈的大小。如果使用 Sun 的 javac 编译器的话，addTwoTypes()的局部变量区需要 3 个字长，操作数栈需要 4 个字长(帧数据区的大小依赖于具体的实现)。虚拟机紧接着从堆中为这个方法分配足够大的栈帧。然后从方法 addAndPrint()的操作数栈中弹出 double 参数和 int 参数，并把它们分别放在 addTwoTypes()的局部变量区中索引为 1 和 0 的位置。

当 addTwoTypes()返回时，它首先把类型为 double 的返回值压入自己的操作数栈里。紧接着虚拟机使用帧数据区中的信息找到调用者(即 addAndPrint())的栈帧，然后将返回值压入 addAndPrint()的操作数栈中并释放方法 addTwoPrint()的栈帧所占用的内存，然后虚拟机把 addTwoTypes()的栈帧作为当前帧，从调用执行的下一条指令开始继续执行方法 addAndPrint()。

其实还有另外一种虚拟机实现执行同一方法的 Java 栈方式，它的栈帧不是从堆中单独分配，而是从一个连续的栈中分配，因而这种方式允许相邻方法的栈帧可以相互重叠。这里调用者的操作数栈部分(它包含要传给被调用者的参数)就成为了被调用者的局部变量区的底层。在这个例子中，addAndPrint()的整个操作数栈刚好成为 addTwoPrint()的整个局部变量区。这种方式不仅节省了内存空间，因为发起调用的方法和被调用的方法用相同的内存保存参数，而且也节省了时间，因为虚拟机不再需要费时地把参数从一个栈帧拷贝到另一个栈帧了。

注意当前帧的操作数栈总是在 Java 栈的“顶”部，尽管这样可能可以更好地说明连续内存的实现，但不管 Java 栈是如何实现的，对操作数栈的压入(或者从栈中弹出)总是在当前帧执行的，这样在当前帧的操作数栈中压入一个值也可以看作是往整个 Java 栈压入一个值。

Java 栈还有一些其他的实现方式，但基本上都是上述两种情形的混合。比如虚拟机可以在线程启动时就从栈中分出一大段空间，之后只要还在这段连续的空间里，虚拟机都可以采用上面介绍的重叠方法。如果栈生长超过了这段连续空间，虚拟机可以再从堆中分配另一段空间。如果发起调用的方法的栈帧位于旧的那段空间中，而被调用的方法的栈帧位于新的那段空间，就需要把它们链接起来，在新的空间段中，虚拟机又可以继续使用连续内存方法。

5.3.9 本地方法栈

前面提到的所有运行时数据区都是在 Java 虚拟机规范中明确定义的，除此之外，对于一个运行中的 Java 程序而言，它还可能会用到一些跟本地方法相关的数据区。当某个线程调用一个本地方法时，它就进入了一个全新的并且不再受虚拟机限制的世界。本地方法可

以通过本地方法接口来访问虚拟机的运行时数据区，但不止于此，它还可以做任何它想做的事情。比如它甚至可以直接使用本地处理器中的寄存器，或者直接从本地内存的堆中分配任意数量的内存等。总之，它和虚拟机拥有同样的权限(或者说能力)。

本地方法本质上是依赖于实现的，虚拟机实现的设计者们可以自由地决定使用怎样的机制来让 Java 程序调用本地方法。

任何本地方法接口都会使用某种本地方法栈。当线程调用 Java 方法时，虚拟机会创建一个新的栈帧并压入 Java 栈。然而当它调用的是本地方法时，虚拟机会保持 Java 栈不变，不再在线程的 Java 栈中压入新的帧，虚拟机只是简单地动态链接并直接调用指定的本地方法。可以把这看作是虚拟机利用本地方法来动态扩展自己。就如共 Java 虚拟机是的实现在按照其中运行的 Java 程序的吩咐，调用属于虚拟机内部的另一(动态链接的)方法。

如果某个虚拟机实现的本地方法接口是使用 C 连接模型的话，那么它的本地方法栈就是 C 栈。我们知道，当 C 程序调用一个 C 函数时，其栈操作都是确定的。传递给该函数的参数以某个确定的顺序压入栈，它的返回值也以确定的方式传回调用者。同样，这就是该虚拟机实现中本地方法栈的行为。

很可能本地方法接口需要回调 Java 虚拟机中的 Java 方法(这也是由设计者决定的)，在这种情形下，该线程就会保存本地方法栈的状态并进入到另一个 Java 栈。

就像其他运行时内存区一样，本地方法栈占用的内存区也不必是固定大小的，它可以根据需要动态扩展或者收缩。某些实现也允许用户或者程序员指定该内存区的初始大小、最大值和最小值。

5.3.10　执行引擎

任何 Java 虚拟机实现的核心都是它的执行引擎。在 Java 虚拟机规范中，执行引擎的行为使用指令集来定义。对于每条指令、规范都详细规定了当实现执行到该指令时应该处理什么，但是却对如何处理言之甚少。在前面的章节中提到过，实现的设计者有权决定如何执行字节码，实现可以采取解释、即时编译或者直接用芯片上的指令执行，还可以是它们的混合，或任何你能想到的新技术。

和本章开头提到的对“Java 虚拟机”这个术语有三种不同的理解一样，“执行引擎”这个术语也可以有如下三种解释：

- 抽象的规范。
- 具体的实现。
- 正在运行的实力。

抽象规范使用指令集规定了执行引擎的行为。具体实现可能使用多种不同的技术——包括软件方面、硬件方面或数种技术的集合。作为运行时实例的执行引擎就是一个线程。

运行中 Java 程序的每一个线程都是一个独立的虚拟机执行引擎的实例。从线程生命周期的开始到结束，它要么在执行字节码，要么在执行本地方法。一个线程可能通过解释或者使用芯片级指令直接执行字节码，或者间接通过即时编译器质性编译过的本地代码。Java 虚拟机的实现可能用一些对用户程序不可见的线程，比如垃圾收集器。这样的线程不需要是实现的执行引擎的实例，所有属于用户运行程序的线程，都是在实际工作的执行

引擎。

指令集方法的字节码流是由 Java 虚拟机的指令序列构成的，每一条指令包含一个单字节的操作码，后面跟随 0 个或多个操作数。操作码表明需要执行的操作；操作数向 Java 虚拟机提供执行操作码需要的额外信息。操作码本身就已经规定了它是否需要跟随操作数，以及如果有操作数的话，它是什么形式的。很多 Java 虚拟机的指令不包含操作数，仅仅是由一个操作码字节构成的。根据这些操作码的需要，虚拟机可能除了跟随操作码的操作数之外，还需要从另外一些存储区域得到操作数。当虚拟机执行一条指令的时候，可能使用当前常量池中的项，当前帧的局部变量中的值，或者位于当前帧操作数栈顶端的值。

抽象的执行引擎每次执行一条字节码指令。Java 虚拟机中运行的程序的每个线程(执行引擎实例)都执行这个操作。执行引擎取得操作码，如果操作码有操作数，取得它的操作数。它执行操作码和跟随的操作数规定的动作，然后再取得下一个操作码。这个执行字节码的过程在线程完成前将一直持续，通过从它的初始方法返回，或者没有捕获抛出的异常都可以标志着线程的完成。

执行引擎会不时遇到请求本地方法调用的指令。在这个时候，虚拟机负责试着发起这个本地方法调用。如果本地方法返回了(假设是正常返回，而不是抛出了一个异常)，执行引擎会继续执行字节码流中的下一条指令。

可以这样来看，本地方法是 Java 虚拟机指令集的一种可编程扩展。如果一条指令请求一个对本地方法的调用，执行引擎就会调用这个本地方法。运行这个本地方法就是 Java 虚拟机对这条指令的执行。当本地方法返回了，虚拟机继续执行下一条指令。如果本地方法异常中止了(抛出了一个异常)，虚拟机就按照好比是这条指令抛出这个异常一样的步骤来处理这个异常。

执行一条指令包含的任务之一就是决定下一条要执行的是什么指令。执行引擎决定下一个操作码时有三种方法。很多指令的下一个操作码就是在当前操作码和操作数之后紧跟的那个字节(如果字节码流里面还有的话)。另外一些指令，比如 goto 或者 return，执行引擎决定下一个操作码时把它当作当前执行指令的一部分。假若一条指令抛出了一个异常，那么执行引擎将搜索合适的 catch 子句，以决定下一个执行的操作码是什么。

有些指令可以抛出异常。比如，athrow 指令就明确地抛出一个异常。这条指令就是 Java 源代码中的 throw 语句的编译后形式。每当执行一条 athrow 指令的时候，它都将抛出一个异常。其他抛出异常的指令都只有在满足某些特定条件的时候才抛出异常。比如，假若 Java 迅即发现程序试图用 0 除一个整数，它就会抛出一个 ArithmeticException 异常。这只有在执行 4 条特定的除法(idev,idiv,irem 和 lrem)的时候，或者计算 int 或者 long 的余数的时候，才可能发生。

Java 虚拟机指令集的每种操作码都有助记符。使用典型的汇编语言风格，Java 字节码流可以用助记符跟着(可选的)操作数值来表示。

方法的字节码流和助记符的例子如下，请读者重点考虑这个类里的 doMathForever() 方法。

```
public class Act {
    public static void doMathForever(){
        int i = 0;
```

```
            for (;;) {
                i += 1;
                i+= 2;
            }
        }
}
```

方法 doMathForever()的字节码流可以被反汇编成下面的助记符。Java 虚拟机规范中没有定义正式的方法字节码的助记符的语法。下面显示的代码说明了本节所采用的用助记符表示字节码的方式。左边的列表表示每条指令开始时从字节码的开头开始算起的每天指令的字节偏移量；中间的列表表示指令和它的操作数；右边的列包含注释，用双斜杠隔开，如同 Java 源代码的格式。

这种表示助记符的方式和 Sun 的 Java 2 SDK 里的 javap 程序的输出很相似。使用 javap 可以查看任何 class 文件中方法的字节码助记符。请注意跳转地址是按照从方法起始开始算起的偏移量来给出的。Goto 指令导致虚拟机跳转到从方法起始计算的位于偏移量 2 的指令。实际上操作数是负 7，要执行这条指令，虚拟机在当前 PC 寄存器的内容上加上这个操作数。记过就是 iinc 指令的地址：偏移量 2。为了让助记符更加易读，所看到的这条跳转指令后面的操作数已经是经过计算后的结果了。主机符显示的是“goto2”，而不是”goto-7”。

Java 虚拟机指令集关注的中心是操作数栈。一般是把将要使用的值会压入栈中。虽然 Java 虚拟机没有保存任意值的寄存器，但每个方法都有一个局部变量集合。指令集实际的工作方式就是把局部变量当作寄存器，用索引来访问。不过不同于 iinc 指令——它可以直接增加一个局部变量的值，要使用保存在局部变量中的值之前，必须先将它压入操作数栈。我们可以看看下面一组指令在执行引擎中执行的过程：

```
//JVM 助记指令代码
iload 0    // 把存储在局部变量区中索引为 0 的整数压入操作数栈。
iload 1    // 把存储在局部变量区中索引为 1 的整数压入操作数栈。
iadd         // 从操作数栈中弹出两个整数相加，在将结果压入操作数栈。
istore 2   // 从操作数栈中弹出结果。
//JVM 助记指令代码。
iload 0    // 把存储在局部变量区中索引为 0 的整数压入操作数栈。
iload 1    // 把存储在局部变量区中索引为 1 的整数压入操作数栈。
iadd         // 从操作数栈中弹出两个整数相加，在将结果压入操作数栈。
istore_2   // 从操作数栈中弹出结果。
```

由此可见，上面的指令反复用到了 Java 栈中的某一个方法栈帧。实际上执行引擎运行 Java 字节码指令很多时候都是在不停地操作 Java 栈，也有的时候需要在堆中开辟对象以及运行系统的本地指令等。但是 Java 栈的操作要比堆中的操作要快得多，因此反复开辟对象是非常耗时的。这也是为什么 Java 程序优化的时候，尽量减少 new 对象。

举例来说，用一个局部变量除另外一个，虚拟机必须把它们都压入栈，执行除法，然后把结果重新保存到局部变量。要把数组元素或对象的字段保存到局部变量中，虚拟机必须先把值压入栈，然后保存到局部变量中去。要把保存在局部变量中的值赋予数组元素或者对象字段，虚拟机必须按照相反的步骤操作。它首先必须把局部变量的值压入栈，然后从栈中弹出，再放入位于堆上的数组元素或对象字段中。

Java 虚拟机指令集的设计遵循几个不同的目标，但它们之间是有冲突的。这几个目标就是本书前面所描述的整个 Java 体系结构的目的所在：平台无关性、网络移动性以及安全性。

平台无关性是影响指令集设计的最大因素。指令集的这种以栈为中心、而非以寄存器为中心的设计方法，使得在那些只有很少的寄存器，或者寄存器很没有规律的机器上实现 Java 更便利，Intel80X86 就是一个例子。由于指令集具有这种以栈为中心的特征，所以在很多平台体系结构上都很容易实现 Java 虚拟机。

Java 以栈为中心设计指令集的另一个动机是，编译器一般采用以栈为基础的结构向连接器或优化器传递编译的中间结果。Javaclass 文件在很多方面都和 C 编译器产生的 o 文件或者.obj 文件很相似，实际上它表示了一种 Java 程序的中间编译结果形式。对于 Java 的情况来，虚拟机是作为(动态)连接器使用的，也可以作为优化器。在 Java 虚拟机指令集设计中，以栈为中心的体系结构可以将运行时进行的优化工作与执行即时编译或者自适应优化的执行引擎结合起来。

在第 4 章中讲过，设计中一个主要考虑因素是 class 文件的紧凑性。紧凑性对于提高在网络上传递 class 文件的速度是很重要的。在 class 文件中保存的字节码，除了两个处理表跳转的指令之外，都是按照字节对齐的。操作码的总数很小，所以操作码可以只占据一个字节。这种设计策略有助于 class 文件的紧凑，但却是以可能影响程序运行的性能为代价的。某些 Java 虚拟机实现，特别是那些在芯片上执行字节码的实现，但字节的操作码可能使得一些可以提高性能的优化无法实现。同样，假若字节码流是以字对齐而非字节对齐的话，某些实现可能会得到更好的性能。实现可以重新对齐字节码流，或者在装载类的时候把操作码转换成更加有效的形式。字节码在 class 文件中是按字节对齐的，在抽象方法区和执行引擎的规范中也是这么规定的。不同的具体实现可以用它们喜欢的任何形式保存装载后的字节码流。

指导指令集设计的另一个目标就是进行字节码验证的能力，特别是使用数据流分析器进行的一次性验证。Java 的安全框架需要这种验证能力。在装载字节码的时候使用数据流分析器进行一次性验证，而非在执行每条指令的时候进行验证，这样做有助于提高执行速度。在指令集中体现这个目标的表现之一，就是绝大部分操作码都指明了它们需要操作的类型。

比如说，不是简单地采用一条指令(该指令从操作数栈中取出一个字并保存到局部变量中)，Java 虚拟机的指令集而是采用两条指令。一条指令是 istore——弹出并保存 int 类型；另一条指令是 fstore——弹出并保存 float 类型。在执行的时候这两条指令所完成的功能是完全一致的：弹出一个字并保存。要区分弹出并保存的到底是 int 类型还是 float 类型，只对验证过程有重要作用。

对于某些指令，虚拟机需要知道被操作的类型，以决定如果执行操作。比如，Java 虚拟机支持两种把两个字加起来并得到一个结果字的操作。一种是把字当作 int 处理，另一种是把字当作 float 处理。这两条指令的区别在于方便验证，同时也告诉虚拟机需要的是整数操作，还是浮点数操作。

有一些指令可以操作任何类型。比如说 dup 指令，不管栈顶的字是什么类型都可以复

制它。还有一些指令不操作有类型的值，比如 goto。但是大部分指令都操作特定的类型。这种“有类型”的指令，可以使用助记符通过一个字符前缀来表明它们操作的类型。表 5-1 中列举了不同类型的前缀。有一些指令不包含前缀，比如 arraylength 或 instanceof，因为他们的类型是再明显不过的。arraylength 操作码需要一个数组引用。instanceof 操作码需要一个对象引用。

表 5-1 字节码助记符的前缀

类 型	代 码	举 例	描 述
byte	b	baload	从数组装载 byte 类型
short	s	sastore	将 short 类型存入数组中
int	i	iload_1	从局部变量 1 中装载 int 类型
long	l	lcmp	比较 long 类型值
char	c	i2c	把 int 类型数据转换为 char 类型
float	f	fload	从局部变量中装载 float 类型
double	d	dconst_1	将 double 类型常量 1.0 压入栈
reference	a	aaload	从数组装载引用类型

操作数栈中的数值必须按照适合它们类型的方式使用。比如说压入 4 个 int，但却把它们当作两个 long 来做加法，这是非法的。把一个 float 值从局部变量压入操作数栈，然后把它作为 int 保存到堆中的数组中去，这也是非法的。从一个位于堆中的对象字段压入一个 double 值，然后把栈中最顶端的两个字作为类型引用保存到局部变量，这也是非法的。Java 编译器所坚持的强类型规则对 Java 虚拟机实现同样也是适用的。

当执行那些与类型无关的一般性栈操作指令时，实现也必须遵守一些规则。前面讲过，不管是什么类型，dup 指令压入栈中顶端那个字的拷贝。这条指令可以用在任何占据一个字的值类型上，如 int,flot、引用或其他的返回类型。但是，如果栈顶包含的是 long 或者 double 类型，它们占据了两个连续的栈空间，这时使用 dup 就是非法的。位于栈顶的 long 或者 double 需要用 dup2 指令复制两个字，在操作数栈中压入栈顶的两个字的拷贝。一般性指令不能用来切割双字值。

为了使指令集足够小，用单字节表示每一个操作码，但并不是在所有类型上都支持所有的操作。很多操作对 byte、short 和 char 都不支持。这些类型在从堆或者方法区转移到栈帧的时候被转换成 int，当它们被当作 int 来进行操作，然后在操作完成后重新保存到堆或方法去的时候，再转换为 byte、short 或者 char。

表 5-2 展示了 Java 虚拟机中保存的每个类型所对应的计算类型。这里，保存类型是堆中类型值所体现的形式。保存类型对应 Java 源代码中变量的类型。计算类型是这些类型在 Java 栈帧中体现的形式。

表 5-2 Java 虚拟机中的保存类型和计算类型

保存类型	堆或者方法区中的最小比特数	计算类型	Java 栈帧中的字长
byte	8	int	1
short	16	int	1

续表

保存类型	堆或者方法区中的最小比特数	计算类型	Java 栈帧中的字长
int	32	int	1
long	64	long	2
char	16	int	1
float	32	float	1
double	64	double	2
reference	32	reference	1

基本类型终结符如表 5-3 所示。

表 5-3　基本类型终结符

终 结 符	类　型
B	byte
C	char
D	double
F	float
I	int
J	long
S	short
Z	boolean

Java 虚拟机实现必须以某种方法确保数值是被对应其类型的指令所操作的。实现可以在类验证过程中就预先验证字节码，或者在执行的时候验证，或者采用前两种验证方式的混合方式。

在具体实现时可以使用多种执行技术：解释、即时编译、自适应优化、芯片级直接执行。关于执行技术要记住的最主要的一点就是，实现可以自由选择任何技术来执行字节码，只要它遵守 Java 虚拟机指令集的定义。

最有意义也是最迅速的执行技术之一是自适应优化。自适应优化已经在几种现有的 Java 虚拟机实现中使用了，如 Sun 的 Hotspot 虚拟机。它们都从早期虚拟机实现所使用的技术中得到了很多借鉴。最初的虚拟机每次解释一条字节码；第二代虚拟机加入了即时编译器，在第一次执行方法的时候先编译本地代码，然后执行这段本地代码。也就是说，不管什么时候调用方法，总是执行本地代码。自适应优化器搜集那些只在运行时才有效的信息，试图以某种方式把字节码解释和编译成本地代码结合起来，以得到最优化的性能。

自适应优化的虚拟机开始的时候对所有的代码都是解释运行，但是它会监视代码的执行情况。大多数程序花费 80%～90%的时间用来执行 10%～20%的代码，它们占整个执行时间的 80%～90%。

当自适应优化的虚拟机判断出某个特定的方法是瓶颈的时候，它启动一个后台线程，把字节码编译成本地代码，非常仔细地优化这些本地代码。同时，程序仍然通过解释来执

行字节码。因为程序没有中途挂起，并且只编译和优化那些“热区”虚拟机可以比传统的即时编译更注重优化性能。

自适应优化技术使程序最终能把原来占 80%～90%运行时间的代码变为极度优化的，静态链接的 C++本地代码，而使用的总内存数并不比全部解释 Java 程序大多少。换句话说，就是更快了。自适应优化的虚拟机可以保留原来的字节码，等待方法从热区移出(程序的热区在执行的过程中可能会转移)。当方法变得不再是热区的时候，取消那些编译过的代码，重新开始解释执行那些字节码。

读者可能会注意到，自适应优化方法令 Java 程序运行得更快，它采取的办法和程序员用来提高程序性能的方法是很相似的。不同于通常的即时编译虚拟机，自适应优化的虚拟机并不进行“过早的优化”。自适应优化的虚拟机通过解释执行字节码开始，当程序运行的时候，虚拟机统计程序，找到程序的热区——就是那 10%～20%的代码，它们花费了 80%～90%的运行时间。就如同一个优秀的程序员那样，自适应优化的虚拟机只对那些对性能产生重大影响的代码进行仔细优化。

但是自适应优化的情况不止这些，它还有另外的着眼点。自适应优化器可以在运行时根据 Java 程序的特征进行微调——特别是对“设计良好”的 Java 程序。根据 JavaSoftHotspot 的经历 DavidGriswold 的说法，“Java 比 C++更加面向对象。你可以测量它，可以发现方法调用的频度、动态派发的频度，等等。这些频度要比 C++中高的多”。现在，在一个设计良好的 Java 程序中，这种方法调用和动态派发的频度更加高了，因为 Java 程序良好设计的尺度之一就是高效率、高产出的设计——换句话就是，使方法和对象更紧凑及内聚性更高。

这些 Java 程序的运行时特征，就是方法调用和动态派发的高频度发生，它们从两个方面影响性能。首先，每次动态派发都会产生相关的管理费用，其次，更重要的是方法调用降低了编译器优化的有效性。

方法调用会使优化器的有效性降低，因为优化器在不同的方法调用间不能够有效地工作，因此优化器在方法调用的时候就无法专注于代码了。方法调用频度越高，方法调用之间可以用来优化的代码就越少，优化器就变得越低效。

这个问题的标准解决方案就是内嵌——把被调用方法的方法体直接拷贝到发起调用的方法中。内嵌调出了方法调用，因此可以让优化器处理更多的代码。这可能令优化器工作更有效，代价就需要更多的运行时内存。

麻烦之处在于，在面向对象的语言(比如 Java 和 C++)中实现内嵌，要比非面向对象的语言(比如 C)更加困难，因为面向对象语言使用了动态派发。在 Java 中比在 C++中更加严重，因为 Java 的方法调用和动态派发的频度要比 C++高得多。

一个 C 程序的标准优化静态编译器可以直接使用内嵌，因为每一个函数调用都有一个函数实现。对于面向对象语言来说，内嵌就变得复杂了，因为动态方法派发以一个函数调用可能有多个函数实现(方法)。换句话说，虚拟机运行时根据方法调用的对象类，可能会有很多不同的方法实现可供选择。

内嵌一个动态派发的方法调用，一种解决办法就是把所有可能在运行时被选择的方法实现都内嵌进去，这种思路的问题在于，如果有很多方法实现，就会让优化后的代码变得

非常大。

自适应编译比静态编译的优点在于，因为它是在运行时工作的，它可以使用静态编译器所无法得到的信息。比如说，对于一个特定的方法调用，就算有 30 个可能的方法实现，运行时可能只会有其中的两个被调用。自适应方法就可以只把这两个方法内嵌，有效地减少过了优化后的代码大小。

线程 Java 虚拟机规范定义了线程模型，这个模型的目标是要有助于在很多体系结构上都实现它。Java 线程模型的一个目标就是使实现的设计者，在可能的情况下使用本地线程。否则，设计者可以在他们的虚拟机实现内部实现线程机制。在一台多处理器的主机上使用本地线程的好处就是，Java 程序不同的线程可以在不同的处理器上并行工作。

Java 线程模型的折中之一就是优先级的规范考虑最小公分母问题。Java 线程可以运行于 10 个优先级的任何一个。级别 1 是优先级最低的，而级别 10 是最高的。如果设计者使用本地线程，他可以用合适的方法把 10 个 Java 优先级映射到机器本地的优先级上。Java 虚拟机规范对于不同优先级别的线程行为，只规定了所有高优先级的线程会得到大多数的 CPU 时间。低级别的线程在级别高的线程没有被阻塞的时候也可能得到 CPU 时间，但是这没有任何保证。

规范没有假设不同优先级的线程采用时间分片方式。因为并不是所有的体系结构都采用时间片(在这里，时间分片的含义是：就算没有线程被阻塞，所有优先级的所有线程都会保证得到一些 CPU 时间)。就算在那些采用时间片的体系结构上，用来分配时间片给不同优先级线程的算法也存在非常大的差异。

我们知道，程序的正确运行不能依靠时间分片。只有在向 Java 虚拟机给出提示，某个线程应该比其他线程使用更多的时间，这时候才使用线程优先级。要协调多线程之间的活动，应该使用同步。

任何 Java 虚拟机的线程实现都必须支持同步的两个方面：对象锁定、线程等待和通知。对象锁定使独立运行的线程访问共享数据的时候互斥。线程等待和通知使得线程为了达到同一个目标而互相协同工作。运行中的程序通过 Java 虚拟机指令集来访问上锁机制，还通过 Object 类的 wait()方法、notify()方法和 notifyAll()方法来访问线程等待和通知机制。

在 Java 虚拟机规范中，Java 线程的行为是通过术语——变量、内存和工作内存——来定义的。每一个 Java 虚拟机实例都有一个主存，用于保存所有的程序变量(对象的实例变量、数组的元素以及类变量)。每一个线程都有一个工作内存，线程用它保存所使用和赋值的变量的“工作拷贝”，局部变量和参数，因为他们是每个线程私有的，可以从逻辑上看成是工作内存或者主存的一部分。

Java 虚拟机规范定义了许多规则，用来管理线程和主存之间的额低层交互行为。比如，一条规则声明：所有对基本类型的操作，除了某些对 long 类型和 double 类型的操作之外，都必须是源自级的。再比如，如果两个线程竞争，对一个 int 变量写了不同的两个值，就算不存在同步，变量最终会采用二者之一。变量不会包含一个不正确的值。或者说，如果一个线程赢得了竞争，把它要写的值先写入到了变量。但失败的那个线程也可以重写那个变量，覆盖那个以为自己“胜利”的线程所写入的值。

这条规则也有例外情况，即任何没有生命为 volatile 的 long 或者 double 变量。某些实现可能把它们作为两个原子性的 32 位值对待，而非一个原子性的 64 位值。比如说，把一个非 volatile 的 long 保存到内存，可能是两次 32 位的写操作。这种对于 long 和 double 的非原子操作可能导致两个竞争性的线程在试图写入不同的值到一个 long 或者 double 变量时，最终得到的是一个不正确的结果。

虽然实现的设计者不是必须对非 volatile 的 long 和 double 进行原子处理，但 Java 虚拟机规范鼓励他们这么做。这种对 long 和 double 的非源自操作，对那条“对所有基本类型的操作都必须是原子级的”的规则而言，就是一个例外，这个例外的目的是，如果处理器不有效的支持和内存交换 64 位的值，线程模型也能经济的实现。将来这个例外可能被终止。然而现在，Java 程序员必须确保通过同步来操作共享的 long 和 double。

基本上，管理低层线程行为的规则，规定了一个线程何时可以做及何时必须做以下的事情：

- 把变量的值从主存复制到它的工作内存。
- 把值从它的工作内存写回到主存。

在特定条件下，规则指定了精确的和可预言的读写内存的顺序。然而另外一些条件下，规则没有规定任何顺序。规则，是设计来让 Java 程序员利用可以预期的行为建立多线程程序，而给实现的设计者更多的灵活性。这种灵活性使 Java 虚拟机实现的设计者从标准硬件和软件技术中得到好处，它们可以提高多线程程序的性能。

所有管理线程行为的低层规则的高层含义是：如果访问某个没有被同步的变量，允许线程用任何顺序来更新主存。不使用同步，多线程程序可能在某些 Java 虚拟机实现上表现出令人惊讶的行为。但是通过正确的使用同步，可以创建多线程的 Java 程序，它们按照可以预期的方式，可以在任何 Java 虚拟机上工作。

5.3.11　本地方法接口

并不强求 Java 虚拟机实现支持任何特定的本地方法接口。有些实现可以根本不支持本地方法接口，还有一些可能支持少数几个，每一个对应一种不同的需求。

Sun 的 Java 本地接口，或者成为 JNI，是为可移植性准备的。JNI 设计的可以被任何 Java 虚拟机实现支持，而不管它们使用何种垃圾洲际或者对象表示技术。这样它能使开发者在一个特定的主机平台上，把同样的(与 JNI 兼容的)本地方法二进制形式连接到任何支持 JNI 的虚拟机实现上。

实现设计者可以选择创建一些私有的本地方法接口，扩展或者取代 JNI。为了实现可移植性，JNI 在指针和指针之间，指针和方法之间使用了很多间接方法。为了得到最好的性能，实现设计者可以提供他们自己的低层本地方法接口，以便和他们所使用的特定实现结构能更加紧密地结合。设计者也可以提供比 JNI 更能高层的本地方法接口，比如把 Java 对象加入到一种组建软件模型中。

为了做好工作，本地方法必须能够和 Java 虚拟机实例的某些内部状态有某种程度的交互。比如，本地方法接口允许本地方法完成下列部分或全部工作：

- 传递或返回数据。

- 操作实例变量或者调用使用垃圾收集的堆中的对象的方法。
- 操作类变量或者调用类方法。
- 操作数组。
- 对堆中的对象加锁，以便被当前线程独占使用。
- 在使用垃圾收集的堆中创建新的对象。
- 装载新的类。
- 抛出新的异常。
- 捕获本地方法调用的 Java 方法抛出的异常。
- 捕获虚拟机抛出的异步异常。
- 指示垃圾收集器某个对象不再需要。

设计一个提供这些服务的本地方法接口是非常复杂的，需要确认垃圾收集器没有释放那些正在被本地方法使用的对象。如果实现的垃圾收集器为了减少堆碎片移动了一个对象，本地方法设计必须保证下面二者之一：

- 当对象的引用被传递给了一个本地方法之后，它可以移动。
- 任何其引用传递给了本地方法的对象都被钉住，直到本地方法返回，或者它表明自己已经完成了对象的操作。

由此可见，本地方法接口和 Java 虚拟机内部工作纠缠在了一起。

5.4　Java 对象池技术的原理及其实现

Java 对象的生命周期大致包括三个阶段：对象的创建、对象的使用、对象的清除。因此，对象的生命周期长度可用如下的表达式表示：T = T1 + T2 +T3。其中 T1 表示对象的创建时间，T2 表示对象的使用时间，而 T3 则表示其清除时间。由此，我们可以看出，只有 T2 是真正有效的时间，而 T1、T3 则是对象本身的开销。下面再看看 T1、T3 在对象的整个生命周期中所占的比例。

众所周知，Java 对象是通过构造函数来创建的，在这一过程中，该构造函数链中的所有构造函数也都会被自动调用。另外在默认情况下，当调用类的构造函数时，Java 会把变量初始化成确定的值：所有的对象被设置成 null，整数变量(byte、short、int、long)设置成 0，float 和 double 变量设置成 0.0，逻辑值设置成 false。所以用 new 关键字来新建一个对象的时间开销是很大的，如表 5-4 所示。

表 5-4　一些操作所耗费时间的对照表

运算操作	示　例	标准化时间
本地赋值	i n	1 0
实例赋值	this i n	1.2
方法调用	Funct()	5.9
新建对象	New Object()	980
新建数组	New int[10]	3100

从表 5-4 中可以看出，新建一个对象需要 980 个单位的时间，是本地赋值时间的 980 倍，是方法调用时间的 166 倍，而若新建一个数组所花费的时间就更多了。再看清除对象的过程。我们知道，Java 语言的一个优势是 Java 程序员无须再像 C/C++程序员那样，显式地释放对象，而由称为垃圾收集器(Garbage Collector)的自动内存管理系统，定时或在内存凸现出不足时，自动回收垃圾对象所占的内存。凡事有利总也有弊，这虽然为 Java 程序设计者提供了方便，但同时它也带来了较大的性能开销。这种开销包括两方面，首先是对象管理开销，GC 为了能够正确释放对象，它必须监控每一个对象的运行状态，包括对象的申请、引用、被引用、赋值等。其次，在 GC 开始回收“垃圾”对象时，系统会暂停应用程序的执行，而独自占用 CPU。

因此，如果要改善应用程序的性能，一方面应尽量减少创建新对象的次数；同时，还应尽量减少 T1、T3 的时间，而这些均可以通过对象池技术来实现。

5.4.1　对象池技术的基本原理

对象池技术基本原理的核心有两点，分别是缓存和共享，即对于那些被频繁使用的对象，在使用完后，不立即将它们释放，而是将它们缓存起来，以供后续的应用程序重复使用，从而减少创建对象和释放对象的次数，进而改善应用程序的性能。事实上，由于对象池技术将对象限制在一定的数量，也有效地减少了应用程序内存上的开销。

在实现一个对象池时，一般会涉及如下所示的类。

1) 对象池工厂(ObjectPoolFactory)类

该类主要用于管理相同类型和设置的对象池(ObjectPool)，它一般包含如下两个方法：

- createPool：用于创建特定类型和设置的对象池；
- destroyPool：用于释放指定的对象池。

同时为了保证 ObjectPoolFactory 的单一实例，可以采用 Singleton 设计模式，见下述 getInstance 方法的实现：

```
public static ObjectPoolFactory getInstance() {
 if (poolFactory == null) {
  poolFactory = new ObjectPoolFactory();
 }
 return poolFactory;
}
```

2) 参数对象(ParameterObject)类

该类主要用于封装所创建对象池的一些属性参数，如池中可存放对象的数目的最大值(maxCount)、最小值(minCount)等。

3) 对象池(ObjectPool)类

用于管理要被池化对象的借出和归还，并通知 PoolableObjectFactory 完成相应的工作。它一般包含如下两个方法：

- getObject：用于从池中借出对象；
- returnObject：将池化对象返回到池中，并通知所有处于等待状态的线程。

4) 池化对象工厂(PoolableObjectFactory)类

该类主要负责管理池化对象的生命周期，就简单来说，一般包括对象的创建及销毁。该类同 ObjectPoolFactory 一样，也可将其实现为单实例。

5.4.2 通用对象池的实现

对象池的构造和管理可以按照多种方式实现。最灵活的方式是将池化对象的 Class 类型在对象池之外指定，即在 ObjectPoolFactory 类创建对象池时，动态指定该对象池所池化对象的 Class 类型，其实现代码如下：

```
public ObjectPool createPool(ParameterObject paraObj,Class clsType) {
 return new ObjectPool(paraObj, clsType);
}
```

其中，paraObj 参数用于指定对象池的特征属性，clsType 参数则指定了该对象池所存放对象的类型。对象池(ObjectPool)创建以后，下面就是利用它来管理对象了，具体实现如下：

```
public class ObjectPool {
 private ParameterObject paraObj;//该对象池的属性参数对象
 private Class clsType;//该对象池中所存放对象的类型
 private int currentNum = 0; //该对象池当前已创建的对象数目
 private Object currentObj;//该对象池当前可以借出的对象
 private Vector pool;//用于存放对象的池
 public ObjectPool(ParameterObject paraObj, Class clsType) {
  this.paraObj = paraObj;
  this.clsType = clsType;
  pool = new Vector();
 }
 public Object getObject() {
  if (pool.size() <= paraObj.getMinCount()) {
   if (currentNum <= paraObj.getMaxCount()) {
    //如果当前池中无对象可用，而且已创建的对象数目小于所限制的最大值，就利用
    //PoolObjectFactory 创建一个新的对象
    PoolableObjectFactory objFactory =PoolableObjectFactory.getInstance();
    currentObj = objFactory.create Object (clsType);
    currentNum++;
   } else {
    //如果当前池中无对象可用，而且所创建的对象数目已达到所限制的最大值，
    //就只能等待其它线程返回对象到池中
    synchronized (this) {
     try {
      wait();
     } catch (InterruptedException e) {
      System.out.println(e.getMessage());
      e.printStackTrace();
     }
     currentObj = pool.firstElement();
    }
   }
  } else {
   //如果当前池中有可用的对象，就直接从池中取出对象
```

```
    currentObj = pool.firstElement();
   }
   return currentObj;
 }
  public void returnObject(Object obj) {
   // 确保对象具有正确的类型
   if (obj.isInstance(clsType)) {
    pool.addElement(obj);
    synchronized (this) {
     notifyAll();
    }
   } else {
    throw new IllegalArgumentException("该对象池不能存放指定的对象类型");
   }
  }
}
```

从上述代码可以看出，ObjectPool 利用一个 java.util.Vector 作为可扩展的对象池，并通过它的构造函数来指定池化对象的 Class 类型及对象池的一些属性。在有对象返回到对象池时，它将检查对象的类型是否正确。当对象池里不再有可用对象时，它或者等待已被使用的池化对象返回池中，或者创建一个新的对象实例。不过，新对象实例的创建并不在 ObjectPool 类中，而是由 PoolableObjectFactory 类的 createObject 方法来完成的，具体实现如下：

```
public Object createObject(Class clsType) {
 Object obj = null;
 try {
  obj = clsType.newInstance();
 } catch (Exception e) {
  e.printStackTrace();
 }
 return obj;
}
```

这样，通用对象池的实现就算完成了，下面再看看客户端(Client)如何来使用它，假定池化对象的 Class 类型为 StringBuffer：

```
//创建对象池工厂
ObjectPoolFactory poolFactory = ObjectPoolFactory. getInstance ();
//定义所创建对象池的属性
ParameterObject paraObj = new ParameterObject(2,1);
//利用对象池工厂,创建一个存放 StringBuffer 类型对象的对象池
ObjectPool pool = poolFactory.createPool(paraObj,String Buffer.class);
//从池中取出一个 StringBuffer 对象
StringBuffer buffer = (StringBuffer)pool.getObject();
//使用从池中取出的 StringBuffer 对象
buffer.append("hello");
System.out.println(buffer.toString());
. . .
```

由此可以看出，通用对象池使用起来还是很方便的，不仅可以方便地避免频繁创建对象的开销，而且通用程度高。但遗憾的是，由于需要使用大量的类型定型(Cast)操作，再加上一些对 Vector 类的同步操作，使得它在某些情况下对性能的改进非常有限，尤其对那

些创建周期比较短的对象。

5.4.3 专用对象池的实现

由于通用对象池的管理开销比较大，某种程度上抵消了重用对象所带来的大部分优势。为解决该问题，可以采用专用对象池的方法。即对象池所池化对象的 Class 类型不是动态指定的，而是预先就已指定。这样，它在实现上也会较通用对象池简单些，可以不要 ObjectPoolFactory 和 PoolableObjectFactory 类，而将它们的功能直接融合到 ObjectPool 类，具体代码如下。

```
public class ObjectPool {
 private ParameterObject paraObj;//该对象池的属性参数对象
 private int currentNum = 0; //该对象池当前已创建的对象数目
 private StringBuffer currentObj;//该对象池当前可以借出的对象
 private Vector pool;//用于存放对象的池
 public ObjectPool(ParameterObject paraObj) {
  this.paraObj = paraObj;
  pool = new Vector();
 }
 public StringBuffer getObject() {
  if (pool.size() <= paraObj.getMinCount()) {
   if (currentNum <= paraObj.getMaxCount()) {
    currentObj = new StringBuffer();
    currentNum++;
   }
   …
  }
  return currentObj;
 }
 public void returnObject(Object obj) {
  // 确保对象具有正确的类型
  if (StringBuffer.isInstance(obj)) {
   . . .
  }
 }
```

在上述代码中，假设被池化对象的 Class 类型仍然为 StringBuffer，而用省略号表示的地方，表示代码同通用对象池的实现。

在 Java 技术中，恰当地使用对象池技术可以有效地改善应用程序的性能。目前，对象池技术已得到广泛的应用，如对于网络和数据库连接这类重量级的对象，一般都会采用对象池技术。但在使用对象池技术时也要注意如下问题：

- 并非任何情况下都适合采用对象池技术。基本上，只在重复生成某种对象的操作成为影响性能的关键因素的时候，才适合采用对象池技术。而如果进行池化所能带来的性能提高并不重要的话，还是不采用对象池化技术为佳，以保持代码的简明。
- 要根据具体情况正确选择对象池的实现方式。如果是创建一个公用的对象池技术实现包，或需要在程序中动态指定所池化对象的 Class 类型时，才选择通用对象池。而大部分情况下，采用专用对象池就可以了。

第 6 章

详解 Class 文件

Java Class 文件是对 Java 程序二进制文件格式的精确定义。每一个 Java Class 文件都对一个 Java 类或者 Java 接口做出了全面描述。在一个 Class 文件中只能包含一个类或者接口。无论 Java Class 文件在何种系统上产生，无论虚拟机在何种系统上运行，对 Java Class 文件的精确定义，可以使所有 Java 虚拟机都能够正确地读取并解释所有 Java Class 文件。本章将详细讲解 Java 中 Class 文件的基本知识。

6.1 Class 介绍

在《The Java™ Virtual Machine Specification》(第二版)中声明：Java Class 文件由 8 位字节流组成，所有的 16 位、32 位和 64 位数据分别通过读入 2 个、4 个和 8 个字节来构造，多字节数据总是按照 Big-endian 顺序来存放，即高位字节在前(放在低地址)。每个 Class 文件都包含且仅包含一个 Java 类型(类或者接口)。

由此可见，Java Class 文件就是指符合特定格式的字节流组成的二进制文件。这个特定的格式就是指第二节要讨论的 Class 文件格式，亦即在《The Java™ Virtual Machine Specification》中定义的 Class 文件格式。从另一个角度来说，这个特定格式就是指 JVM 能够识别、装载的格式。为什么这么说呢？因为 JVM 在装载 Class 文件时，要进行 Class 文件验证，以保证装载的 Class 文件内容符合正确的内部结构。这个内部结构指的就是这个特定格式，只要是符合这个特定格式的 Class 文件都是合法的、规范的 Class 文件，都是 JVM 能够装载的 Class 文件。

在讲 Class 文件的格式之前，需要先了解和 Class 文件相关的三个概念，具体如下。

1) 数据类型

Java Class 文件的数据用自己定义的一个数据类型集来表示，即 u1、u2、u4，分别用于表示一个无符号类型的、占 1、2、4 个字节的数据。在现实应用中，人们通常把这个数据类型集称之为 Class 文件的基本类型，所以在本书中我们也用基本类型来表示 Java Class 文件的数据。

2) 表

表(table)由项(见本章后面的内容)组成，用于几种 Class 文件结构中。Java Class 文件格式用一个类似于 C 结构的记号编写的伪结构来表示。这个伪结构指的就是这里的表，例如，ClassFile 表就是这种伪结构的一个典型例子，本书中所有的表都是指这种伪结构的表。表的大小是可变的，这是因为它的组成部分项是可变的。注意：这里的可变是针对 Class 层次而言的，即在不同的 Class 文件中该项的大小可能不一样的，但是对于每一个具体的 Class 文件来说，这个项的大小又是一定的，因而这个表的大小也是一定的。

3) 项

描述 Java Class 文件格式的结构的内容称为项(Items)。每个项都有自己的类型和名称。项的类型可能是基本类型，也可能是一个表的名字，这种项都是一些数组项。数组项的每一个元素都是一个表，这个表同顶层的 ClassFile 表一样，也都是一种伪结构，也都是由一些项构成的，而且这些表不一定是同一种格式的，因此，数组项也可以看作一个可变大小的结构流。这些表对于该数组项来说就是子项，当然子项可能还有子项(目前子项的深度最多就两层)。项的名称就是《JVM Spec》(Java 虚拟机规范)中指定的一些名称。另外，项也是有大小的，对于没有子项的项来说，其大小是固定的；对于有子项的项来说，其大小是可变的。在一个具体的 Class 文件中，一个可变项(数组)的大小都会在其前一项中指定，为什么会是这样的呢？因为在 Class 文件中，每个项按规范中定义好的顺序存储在

Class 文件中，相邻的项之间没有任何间隔，连续的项(数组)也是按顺序存储，不进行填充或者对齐，这样可以使 Class 文件紧凑。

虽然 Class 文件与 Java 语言结构相关，但它并不一定必须与 Java 语言相关。如图 6-1 所示，可以使用其他语言来编写程序，然后将其编译为 Class 文件，或者把 Java 程序编译为另一种不同的二进制文件格式。实际上，Java Class 文件的形式能够表示 Java 源代码中无法表达的有效程序，然而，绝大多数 Java 开发者几乎都会选择使用 Class 文件作为程序传给虚拟机的首要方式。

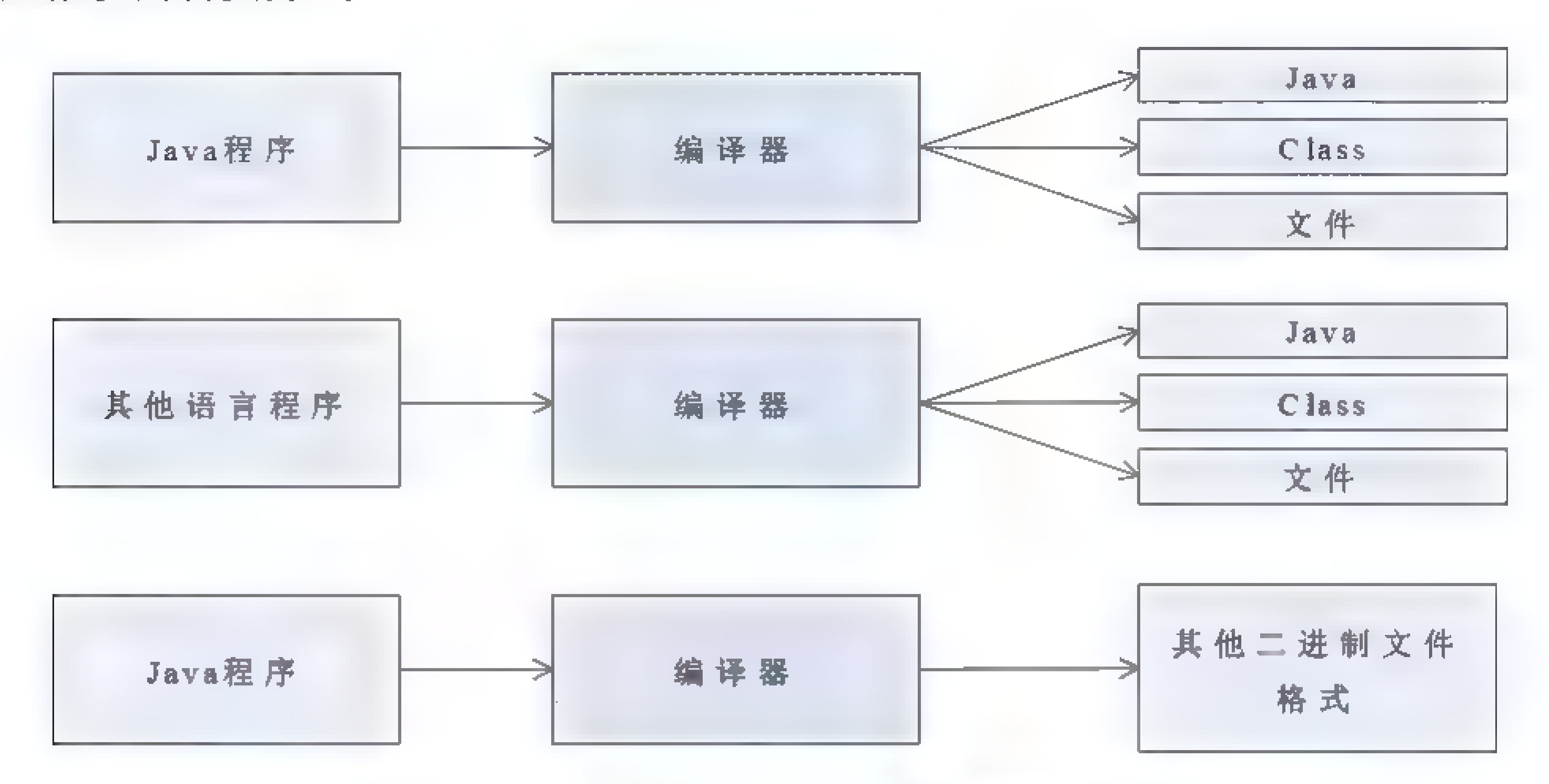


图 6-1　Java 语言与 Class 文件之间排他性的关系

如前所述，Java Class 文件是 8 位字节的二进制流。数据项按顺序存储在 Class 文件中，相邻的项之间没有任何间隔，这样可以使 Class 文件紧凑。占据多个字节空间的项按照高位在前的顺序分为几个连续的字节存放。

和 Java 的类可以包含多个不同的字段，跟方法、方法参数和局部变量等一样，Java Class 文件也能够包含许多不同大小的项。在 Class 文件中，可变长度项的大小和长度位于其实际数据之前。这个特性使得 Class 文件流可以从头到尾被顺序解析，首先读出项的大小，然后读出项的数据。

6.2　Java Class 文件的格式

在接下来的内容中，开始正式解析 Class 文件的格式。要想分析 Class 文件的格式，需要先解析表 ClassFile 的结构，因为这是 Class 文件最外层的结构，也就是 Class 文件的格式。

ClassFile 表的结构如下：

```
ClassFile {
    u4 magic;
    u2 minor_version;
    u2 major_version;
```

```
    u2 constant pool count;
    cp info constant pool[constant pool count-1];
    u2 access flags;
    u2 this Class;
    u2 super Class;
    u2 interfaces count;
    u2 interfaces[interfaces count];
    u2 fields count;
    field info fields[fields count];
    u2 methods count;
    method info methods[methods count];
    u2 attributes count;
    attribute info attributes[attributes count];
}
```

ClassFile 表结构由 16 个不同的项组成，下面列出了主要项的具体说明。

1) magic

每个 class 文件的前 4 个字节被称为它的魔数(magic number):0xCAFEBABE。魔数的作用是可以轻松地分辨出 Java Class 文件和非 Java Class 文件。如果一个文件不是以“0xCAFEBABE”开头，它就肯定不是 Java Class 文件，因为它不符合规范 J。当 Java 还称为“Oak”的时候，这个魔数就已经定下来了，它预示了 Java 这个名字的出现。

2) minor_version 和 major_version

Class 文件的下面 4 个字节包含了次、主版本号。通常只有给定主版本号和一系列次版本号后，Java 虚拟机才能够读取 Class 文件。如果 Class 文件的版本号超出了 Java 虚拟机所能够处理的有效范围，Java 虚拟机将不会处理该 Class 文件。例如，对于 J2SE5.0 版本的虚拟机来说，就不能执行由 J2SE6.0 版本的编译器编译出来的 Class 文件。

3) constant_pool_count

版本号后面的项是 constant_pool_count 即常量池计数项，该项的值必须大于零，它给出该 Class 文件中常量池列表项的元素个数，这个计数项包括了索引为 0 的 constant_pool 表项，但是该表项不出现在 Class 文件的 constant_pool 列表中，因为它被保留为 Java 虚拟机内部实现使用了，因此常量池列表的元素个数 constant_pool_count-1，各个常量池表项的索引值分别为 1 到 constant_pool_count-1。

上面提到的常量池即 constant_pool，常量池列表就是指 constant_pool[ ]，常量池表项即指常量池列表中的某一个具体的表项(元素)。这些常量池表项的可能类型如下：

- 入口类型 CONSTANT_Class：标志值是 7；
- 入口类型 CONSTANT_Fieldref：标志值是 9；
- 入口类型 CONSTANT_Methodref：标志值是 10；
- 入口类型 CONSTANT_InterfaceMethodref：标志值是 11；
- 入口类型 CONSTANT_String：标志值是 8；
- 入口类型 CONSTANT_Integer：标志值是 3；
- 入口类型 CONSTANT_Float：标志值是 4；
- 入口类型 CONSTANT_Long：标志值是 5；
- 入口类型 CONSTANT_Double：标志值是 6；

- 入口类型 CONSTANT_NameAndType：标志值是 12；
- 入口类型 CONSTANT_Utf8：标志值是 1。

4) constant_pool[ ]

constant_pool_count 项下面是 constant_pool[ ]项，即常量池列表，在里面存储了该 ClassFile 结构及其子结构中引用的各种常量，例如，文字字符串、final 变量值、类名和方法名等等。在 Java Class 文件中，常量池表项使用 cp_info 结构来描述，常量池列表就是由 constant_pool_count-1 个连续的、可变长度的 cp_info 表结构构成的 constant_pool[ ]数组。每一个常量池表项都是一个变长结构，其通常格式如下：

```
cp_info {
        u1 tag;
        u1 info[];
    }
```

cp_info 表的 tag 项是一个无符号的 byte 类型值，它表明了 cp_info 表的类型和格式。cp_info 其实只是一个抽象的概念，在 Class 文件中，它表现为一系列具体的、形如 CONSTANT_xxxx_info 的 constant_pool 结构，其具体的格式由 cp_info 表的 tag 项(即第一个字节)来确定。不同的 cp_info 表的 info[]项是不一样的，例如，CONSTANT_Class_info 表的 info[]项是“u2 name_index”，而 CONSTANT_Utf8_info 表的 info[]项是“u2 length; u1 bytes[length];”，很明显，这两个 cp_info 表是不一样的，所以常量池表项的大小是可变的。由于常量池列表中的每个常量池表项的结构是不一样，因此常量池列表的大小也是可变的。在 Class 文件中，常量池列表项是一个可变长度的结构流。

当 cp_info 表中 tag(标志)项的值为 1 时，当前的 cp_info 就是一个 CONSTANT_Utf8_info 表结构，如果 cp_info 表中 tag 项的值为 3，当前的 cp_info 就是一个 CONSTANT_Integer_info 表结构，其他情况类推。

5) access_flags

紧接常量池后的两个字节称为 access_flags，access_flags 项描述了该 Java 类型的一些访问标志信息。例如，访问标志指明文件中定义的是类还是接口；访问标志还定义了在类或接口的声明中，使用了哪些修饰符；类和接口是抽象的还是公共的等等。实际上，access_flags 项的值是 Java 类型声明中使用的访问标志符的掩码(Mask)，这里掩码指的是 access_flags 的值是所有访问标志值的总和，当然，未被使用的标志位在 Class 文件中都被设置为 0。例如，如果 access_flags 的值是 0x0001，则表示该 Java 类型的访问标志符是 ACC_PUBLIC；如果 access_flags 的值是 0x0011，则表示该 Java 类型的访问标志符是 ACC_PUBLIC 和 ACC_FINAL，因为只有这两个标志位的和才可能是 0x0011；其他情况类推。

一个 Java 类型的所有 access_flags 标志符如下：

- ACC_PUBLIC：值是 0x0001，声明为 public，可以从它的包外访问。
- ACC_FINAL：值是 0x0010，声明为 final，不允许有子类。
- ACC_SUPER：值是 0x0020，用 invokespecial 指令处理超类的调用。
- ACC_INTERFACE：值是 0x0200，表明是一个接口，而不是一个类。
- ACC_ABSTRACT：值是 0x0400，声明为 abstract，不能被实例化。

需要说明的是，这是针对一个 Java 类型的访问标志符列表，有的标志符只有类可以使用，有的标志符只有接口才可以使用。

6) this_class

接下来的两个字节为 this_class 项，值为一个对常量池表项的索引，即它指向一个常量池表项，而且该常量池表项必须为 CONSTANT_Class_info 表的结构。该表有一个 name_index 项，该项将指向另一个常量池表项，该表项包含了该类或者接口的完全限定名称。

7) super_class

紧接着 this_class 之后的两个字节是 super_class 项，该项必须是对常量池表项的一个有效索引或者值为 0。如果 super_class 项的值为 0，则该 Class 文件必须表示 java.lang.Object 类。如果 super_class 项的值不为 0，则又分为两种情况，若该 Class 文件表示一个类，则 super_class 项必须是对常量池中该类的超类的 CONSTANT_Class_info 表项的索引，这个超类和它的任何超类都不能是一个 final 类；若该 Class 文件表示一个接口，则 super_class 项必须是对常量池中表示 java.lang.Object 类的一个 CONSTANT_Class_info 表项的索引。

8) interfaces_count 和 interfaces[ ]

紧接着 super_class 项后面的两个字节是 interfaces_count 项，此项表示由该类直接实现或者由该接口所扩展的超接口的数量。紧接着 interfaces_count 项后面的是 interfaces 列表项，它包含了由该类直接实现或者由该接口所扩展的超接口的常量池索引，共计 interfaces_count 个索引。interfaces 列表中的常量池索引按照该类型在源代码中给定的从左到右的顺序排列。

9) fields_count 和 fields[ ]

接下来的是 fields_count 项，该项的值给出了 fields 列表项中的 field_info 表结构的数量，即表示了该 Java 类型声明的类变量和实例变量的个数总和。

fields 列表项包含了在该 Java 类型中声明的所有字段的完整描述。fields 列表中的每个 field_info 表项都完整地表示了一个字段的信息，包括该字段的名称、描述符和修饰符等。这些信息有的放在 field_info 表中，例如，修饰符；有的则放在 field_info 表所指向的常量池中，例如，名字和描述符。fields 列表项也是一个变长结构。

在此需要说明的是，只有在该 Java 类型中声明的字段才可能在 fields 列表中列出，fields 列表中不包括从超类或者超接口中继承而来的字段信息。

10) methods_count 和 methods[ ]

在 Class 文件中，紧接着 fields 后面的是对在该 Java 类型中所声明的方法的描述。首先是 methods_count 项，它占两个字节长度，它的值表示对该 Java 类型中声明的所有方法的总计数。methods_count 项后面是 methods 列表项，它由 methods_count 个连续的 method_info 表构成。每个 method_info 表都包含了与一个方法相关的信息，如方法名、描述符(即方法的返回值及参数类型)以及一些其他信息。如果一个方法既非 abstract 也非 native，那么该 method_info 表将包含该方法局部变量所需的栈空间长度、为方法所捕获的异常表、字节码序列以及可选的行号表和局部变量表等信息。

在此需要说明的是，只有在该 Java 类型中显式定义的方法才可能在 fields 列表中列出，fields 列表中不包括从超类或者超接口中继承而来的方法信息。

11) attributes count 和 attributes[ ]

Class 文件中最后的部分是属性(attribute)，它给出了在该 Java 类型中所定义的属性的基本信息。首先是 attributes count 项，它占两个字节长度，它的值表示在后续的 attributes 列表中的 attributes info 表的总个数。每个 attributes info 表的第一项都是对常量池中 CONSTANT Utf8 info 表项的一个索引，该表给出了此属性的名称。

在此需要说明的是，属性有很多种，在 Class 文件中的很多地方都出现了属性这一项，在顶层 ClassFile 表中有 attributes 属性项，在 field_info 表中也有 attributes 属性项，在 method_info 中也有 attributes 属性项，但是它们各有各的功能。权威者曾经为 ClassFile 表结构的 attributes 列表项定义的唯一属性是 SourceFile 属性，为 field_info 表结构的 attributes 列表项定义的唯一属性是 ConstantValue 属性，为 method_info 表结构的 attributes 列表项定义的属性是 Code 属性和 Exceptions 属性。

综上所述，Class 文件格式是一个规范性的格式。此规范指的是上面提到的这些表结构本身的规范性，以及这些表结构之间包含关系的规范性。ClassFile 表就是 Class 文件最外层的结构，也就是说，这就是 Class 文件的格式。而 ClassFile 表又是一些项组成的，这些项的内容都要符合《JVM Spec》中定义的规范。具体来说，如果这个项的类型是基本类型，该项的值要符合规范，例如，magic 项一定要是 0xCAFEBABE，access_flags 项的值一定要是有效的标志值等；若这个项的类型是一个表名，即该项是一个数组项，那么该数组项列表中的每一个表项都要是一个合法的、规范的表，不能是一个规范中没有定义的新表，这就是包含关系的规范性，同样，列表项中的每个表项本身也都要是符合其规范定义的表项，例如常量池列表中的某个 CONSTANT_Class_info 表的 name_index 项不是对一个 CONSTANT_Utf8_info 表结构的索引，那么这个常量池的表项就不是一个合法的表项，因而这个常量池列表项就是不符合规范的，因而整个文件就是不符合规范的。

6.3 常量池的具体结构

在 Java 程序中，有很多的东西是永恒的，不会在运行过程中变化。比如一个类的名字，一个类字段的名字/所属类型，一个类方法的名字/返回类型/参数名与所属类型，一个常量，还有在程序中出现的大量的字面值。比如下面代码中加粗部分显示的内容。

```
public class ClassTest {
      private String itemS ="我们";
      private final int itemI =100 ;
      public void setItemS (String para ){...}
}
```

而这些在 JVM 解释执行程序的时候是非常重要的。那么编译器将源程序编译成 Class 文件后，会用一部分字节分类存储这些永恒不变的红色东西。而这些字节我们就称为常量池。事实上，只有 JVM 加载 Class 后，在方法区中为它们开辟了空间才更像一个“池”。

正如上面所示，一个程序中有很多永恒的加粗东西。每一个都是常量池中的一个常量表(常量项)。而这些常量表之间又有不同，Class 文件共有 11 种常量表，如表 6-1 所示。

表 6-1　Class 文件的常量表

常量表类型	标志值(占 1 byte)	描　述
CONSTANT_Utf8	1	UTF-8 编码的 Unicode 字符串
CONSTANT_Integer	3	int 类型的字面值
CONSTANT_Float	4	float 类型的字面值
CONSTANT_Long	5	long 类型的字面值
CONSTANT_Double	6	double 类型的字面值
CONSTANT_Class	7	对一个类或接口的符号引用
CONSTANT_String	8	String 类型字面值的引用
CONSTANT_Fieldref	9	对一个字段的符号引用
CONSTANT_Methodref	10	对一个类中方法的符号引用
CONSTANT_InterfaceMethodref	11	对一个接口中方法的符号引用
CONSTANT_NameAndType	12	对一个字段或方法的部分符号引用

下面将一个源程序编译成 Class 文件后，对文件中的每一个字节的分析，可以更好地理解 Class 文件的内容以及常量池的组成。Java 代码如下：

```
package hr.test;
//ClassTest 类
public class ClassTest {
   private int itemI=0;  //itemI 类字段
   private static String itemS="我们"; //itemS 类字段
   private final float PI=3.1415926F;  //PI 类字段
   //构造器方法
   public ClassTest(){
   }
   //getItemI 方法
   public int getItemI(){
      return this.itemI;
   }
   //getItemS 方法
   public static String getItemS(){
      return itemS;
   }
   //main 主方法
   public static void main(String[] args) {
      ClassTest ct=new ClassTest();
   }
}
```

接下来开始分析 TestClass.class 文件的字节码，字节顺序从上到下，从左到右。每个字节用一个 0～255 的十进制整数表示。

```
   202 254 186 190   -- 魔数
```

```
    0 0     -- 次版本号
    0 50    -- 主版本号
    0 43       常量池中常量表的数量有 42 个，下面红色括号中的数据表明该常量表所在常量
池中的索引，从索引 1 开始

(1) 7 0 2  -- 对类 ClassTest 的符号引用(7 为标志  02 指向了常量池的索引 2 的位置)

(2) 1 0 17 104 114 47 116 101 115 116 47 67 108 97 115 115 84 101 115
116  -- 类全限定名 hr\test\ClassTest
(3) 7 0 4   -- 对类 Object 的符号引用

(4) 1 0 16 106 97 118 97 47 108 97 110 103 47 79 98 106 101 99 116  --
超类全限定名 java/lang/Object
(5) 1 0 5 105 116 101 109 73   --  第 1 个类字段名 itemI
(6) 1 0 1 73   --  I  第 1 个类字段类型为整型
(7) 1 0 5 105 116 101 109 83   --  第 2 个类字段名 itemS
(8) 1 0 18 76 106 97 118 97 47 108 97 110 103 47 83 116 114 105 110 103
59   --  第 2 个类字段类型的全限定名 Ljava/lang/String
(9) 1 0 2 80 73 -- 第 3 个类字段名 PI
(10) 1 0 1 70  -- 第 3 个类字段类型为 float
(11) 1 0 13 67 111 110 115 116 97 110 116 86 97 108 117 101   ---  第 3 个
类字段为常量 ConstantValue

(12) 4 64 73 15 218  -- 第 3 个类字段 float 字面值,占 4bytes(3.1415926)
(13) 1 0 8 60 99 108 105 110 105 116 62   -- <clinit>  初始化方法名
(14) 1 0 3 40 41 86    -- ()V 方法的返回类型为 void

(15) 1 0 4 67 111 100 101    -- Code
(16) 8 0 17 -- String 字符串字面值(0 17 表示索引 1 7)
(17) 1 0 6 230 136 145 228 187 172   -- "我们"
(18) 9 0 1 0 19  -- 指向 第 2 个 字段的引用(0 1 指向索引 1, 0 19 指向索引 19)
(19) 12 0 7 0 8  --指向 第 2 个 字段的名字和描述符的索引
(20) 1 0 15 76 105 110 101 78 117 109 98 101 114 84 97 98 108 101   --
LineNumberTable
(21) 0 18 76 111 99 97 108 86 97 114 105 97 98 108 101 84 97 98 108 101
-- LocalVariableTable
(22) 1 0 6 60 105 110 105 116 62  -- <init>  表示初始化方法名
(23) 10 0 3 0 24 --  指向父类 Object 的构造器方法，0 3 表示父类名常量表的索引，0
24 表示存放该方法名称和描述符的引用的常量表的索引
(24) 12 0 22 0 14  --  指向方法名和描述符的常量表的索引。0 22 是方法名的常量表索
引，0 14 是描述符的常量表索引
(25) 9 0 1 0 26   -- 指向第 1 个字段的引用，0 1 表示字段所属类型的索引，0 26 表示字
段名和描述符的索引
(26) 12 0 5 0 6   -- 指向第 1 个字段的名字和描述符的索引
(27) 9 0 1 0 28   -- 指向第 3 个字段的引用，0 1 表示字段所属类型的索引，0 28 表示字
段名和描述符的索引
(28) 12 0 9 0 10  -- 指向第 3 个字段的名字和描述符的索引
(29) 1 0 4 116 104 105 115   --  隐含参数符号 this
(30) 1 0 11 76 67 108 97 115 115 84 101 115 116 59   --  LClassTest;
(31) 1 0 8 103 101 116 73 116 101 109 73   - - 方法名 getItemI
(32) 1 0 3 40 41 73   -- ()I  方法描述符:返回类型 int
(33) 1 0 8 103 101 116 73 116 101 109 83   --  方法名 getItemS
(34) 1 0 20 40 41 76 106 97 118 97 47 108 97 110 103 47 83 116 114 105
110 103 59    --- 方法描述符()Ljava/lang/String;
```

```
(35) 1 0 4 109 97 105 110     -- 主方法名main
(36) 1 0 22 40 91 76 106 97 118 97 47 108 97 110 103 47 83 116 114 105
110 103 59 41 86    ---()Ljava/lang/String;)V 主方法中的参数的字符串数组类型名
(37) 10 0 1 0 24    指向当前 ClassTest 类的构造器方法，0 1 表示存放当前类名的常量
表的索引。0 24 是存放方法名和描述符的符号引用的常量表索引。
(38) 1 0 4 97 114 103 115 -- 参数 args
(39) 1 0 19 91 76 106 97 118 97 47 108 97 110 103 47 83 116 114 105 110
103 59   -- 字符串数组 [Ljava/lang/String;
(40) 1 0 2 99 116    ---  对象符号 ct
(41) 1 0 10 83 111 117 114 99 101 70 105 108 101    -- SourceFile
(42) 1 0 14 67 108 97 115 115 84 101 115 116 46 106 97 118 97 -- ClassTest.java
```

上述部分表示常量池字节码区域，具体说明如下。

- 所有的字面值都是存放在常量池中的。特别注意的是“我们”这个字符串常量也是在常量池中的。如果一个程序出现多个“我们”，那么常量池中也只会有一个。另外，也正是因为“我们”存放在常量池中，使得一些字符串的“==”比较变得需要琢磨了——这其实是在解析常量池中的 CONSTANT_String 类型时，采用了拘留字符串，并将其地址放入了 CONSTANT_String_info 入口数据中，以保证整个应用程序应用的字面值相同字符串的唯一。
- ClassTest 并没有任何显示的父类。但在常量池中，我们发现有 Object 的符号常量存在。这说明在 Java 中任何类都直接或间接继承了 Object 的，而 Object 并不需要在代码中显示继承，JVM 会帮我们做到这一点。
- 常量池中有一个隐含参数 this 的符号常量(索引 29)。即使程序中不存在 this，JVM 也会悄悄地设置一个这样的对象。

继续分析此文件的字节码：

```
0 33 ---- access flag 访问标志  public
 0 1  ---- this class  指向当前类的符号引用在常量池中的索引
 0 3  ---- super class
 0 0 ---- inteface count 接口的数量

 0 3   ---  field count 字段的数量

 // 字段 itemI
 0 2  ---- private 修饰符
 0 5  ---- 字段名在常量池中的索引，字段 itemI
 0 6   ---- 字段的描述符(所属类型)在常量池中的索引
 0 0   ---  字段的属性信息表(attribute info)的数量
 // 字段 itemS
 0 10   ----  private static 修饰符
 0 7  ---字段名在常量池中的索引，字段 itemS
 0 8  ---字段的描述符(所属类型)在常量池中的索引
 0 0  ---  字段的属性信息表(attribute info)的数量
  // 字段 PI
 0 18   -- private final 修饰符
 0 9 ---字段名在常量池中的索引，//字段 PI
 0 10 ---字段的描述符(所属类型)在常量池中的索引
 0 1  --- 字段的属性信息表(attribute info)的数量
 0 11   --- 属性名在常量池中的索引。即 ConstantValue
 0 0 0 2 --- 属性所占的字节长度
```

```
    0 12  --- 属性值在常量池中的索引。即常量字面值

    0 5  -- Method count 方法的数量
    //类的静态数据初始化方法<clinit>
    0 8 ---- static 修饰符(所有的初始化方法都是 static 的)
    0 13      在常量池中的索引。初始化方法名<clinit>，该方法直接由 JVM 在特定的时
候调用，并非由字节码生成。
    0 14 --- 在常量池中的索引。返回类型为 void。

    0 1 --- 属性数量
    0 15 -- 属性名在常量池中的索引。即 code
    0 0 0 42 ---  属性所占的字节长度 2
    0 1 0 0 0 0 0 6 18 16 179 0 18 177 0 0 0 2 0 20 0 0 0 10 0 2 0 0 0
5 0 5 0 2 0 21 0 0 0 2 0 0 ---该方法的字节码指令序列和其他信息
    //类的普通实例数据的初始化方法，针对类构造器生成的<init>方法。
    0 1 --- public 修饰符
    0 22 --- 初始化方法名<init>
    0 14 --- 构造器的返回类型为 void
    0 1  --- 属性数量
    0 15 ---  属性名在常量池中的索引。即 Code
    0 0 0 70 -- 属性所占的字节长度 70
    0 2 0 1 0 0 0 16 42 183 0 23 42 3 181 0 25 42 18 12 181 0 27 177 0
0 0 2 0 200 0 0 18 0 4 0 0 0 8 0 4 0 4 0 9 0 6 0 15 0 9 0 21 0 0 0 12 0
10 0 0 16 0 29 0 30 0 0 ---该方法的字节码指令序列和其他信息
    //getItemI 方法
    0 1 --- public 修饰符
    0 31 ---  在常量池中的索引。方法名 getItemI
    0 32 ---  在常量池中的索引。方法返回类型为 int
    0 1 -- 属性数量
    0 15  --- 属性名在常量池中的索引。即 Code
    0 0 0 47 ---  属性所占的字节长度 70
    0 1 0 1 0 0 0 5 42 180 0 25 172 0 0 0 2 0 20 0 0 0 6 0 1 0 0 0 12
0 21 0 0 0 12 0 1 0 0 0 5 0 29 0 30 0 0 ---该方法的字节码指令序列和其他信息
    //getItemS 方法
    0 9 --- public static 修饰符
    0 33 ---  在常量池中的索引。方法名 getItemS
    0 34 - -- 在常量池中的索引。方法返回类型为 String
    0 1 --- 属性数量
    0 15 -- 属性名在常量池中的索引。即 Code
    0 0 0 36 ---  属性所占的字节长度 36
    0 1 0 0 0 0 0 4 178 0 18 176 0 0 0 2 0 20 0 0 0 6 0 1 0 0 0 16 0
21 0 0 0 2 0 0 --该方法的字节码指令序列和其他信息
    //main 方法
    0 9 --- public static 修饰符
    0 35 ---  在常量池中的索引。主方法名 main
    0 36 -- 在常量池中的索引。方法返回类型为 String[]
    0 1 ---  属性数量
    0 15  ---  属性名在常量池中的索引。即 Code
    0 0 0  65 ---  属性所占的字节长度 36
    0 2 0 2 0 0 0 9 187 0 1 89 183 0 37 76 177 0 0 0 2 0 20 0 0 0 10 0
2 0 0 0 20 0 8 0 21 0 21 0 0 0 22 0 2 0 0 0 9 0 38 0 39 0 0 0 8 0 1 0 40
0 30 0 1 0 1 0 41 0 0 0 2 0 42
```

在上述文件中，字段 PI 是浮点型常量，在编译期的字节码中就已经指定好了 PI 的字

面值存储在常量池中的某个索引内。这一点也证实了 Java 中的常量在编译期就已经得到了值，在运行过程中是无法改变的。

继续分析此文件的字节码。

```
    0 5  -- Method count 方法的数量
    //类的静态数据初始化方法<clinit>
    0 8 ---- static 修饰符(所有的初始化方法都是 static 的)
    0 13 --- 在常量池中的索引。初始化方法名<clinit>，该方法直接由 JVM 在特定的时候调用，并非由字节码生成。
    0 14 --- 在常量池中的索引。返回类型为 void。

    0 1 --- 属性数量
    0 15 -- 属性名 在常量池中的索引。即 code
    0 0 0 42 ---  属性所占的字节长度 2
    0 1 0 0 0 0 0 6 18 16 179 0 18 177 0 0 0 2 0 20 0 0 0 10 0 2 0 0 0 5 0 5 0 2 0 21 0 0 0 2 0 0 ---该方法的字节码指令序列和其他信息
    //类的普通实例数据的初始化方法，针对类构造器生成的<init>方法。
    0 1 --- public 修饰符
    0 22 --- 初始化方法名<init>
    0 14 --- 构造器的返回类型为 void
    0 1  --- 属性数量
    0 15 ---  属性名在常量池中的索引。即 Code
    0 0 0 70 -- 属性所占的字节长度 70
    0 2 0 1 0 0 0 16 42 183 0 23 42 3 181 0 25 42 18 12 181 0 27 177 0 0 0 2 0 200 0 0 18 0 4 0 0 0 8 0 4 0 4 0 9 0 6 0 15 0 9 0 21 0 0 0 12 0 10 0 0 16 0 29 0 30 0 0 ---该方法的字节码指令序列和其他信息
    //getItemI 方法
    0 1 --- public 修饰符
    0 31 ---  在常量池中的索引。方法名 getItemI
    0 32 ---  在常量池中的索引。方法返回类型为 int
    0 1 -- 属性数量
    0 15  --- 属性名在常量池中的索引。即 Code
    0 0 0 47 ---  属性所占的字节长度 70
    0 1 0 1 0 0 0 5 42 180 0 25 172 0 0 0 2 0 20 0 0 0 6 0 1 0 0 0 12 0 21 0 0 0 12 0 1 0 0 0 5 0 29 0 30 0 0  ---该方法的字节码指令序列和其他信息
    //getItemS 方法
    0 9 --- public static 修饰符
    0 33 ---  在常量池中的索引。方法名 getItemS
    0 34 - -- 在常量池中的索引。方法返回类型为 String
    0 1 --- 属性数量
    0 15 -- 属性名在常量池中的索引。即 Code
    0 0 0 36 ---  属性所占的字节长度 36
    0 1 0 0 0 0 0 4 178 0 18 176 0 0 0 2 0 20 0 0 0 6 0 1 0 0 0 16 0 21 0 0 0 2 0 0 --该方法的字节码指令序列和其他信息
    //main 方法
    0 9 --- public static 修饰符
    0 35 ---  在常量池中的索引。主方法名 main
    0 36 -- 在常量池中的索引。方法返回类型为 String[]
    0 1 ---  属性数量
    0 15  ---  属性名在常量池中的索引。即 Code
    0 0 0  65 ---  属性所占的字节长度 36
    0 2 0 2 0 0 0 9 187 0 1 89 183 0 37 76 177 0 0 0 2 0 20 0 0 0 10 0 2 0 0 0 20 0 8 0 21 0 21 0 0 0 22 0 2 0 0 0 9 0 38 0 39 0 0 0 8 0 1 0 40
```

```
0 30 0 1 0 1 0 41 0 0 0 2 0 42
```

在上述代码中，主方法 main 是作为 ClassTest 的类方法存在的，在字节码中 main 和其他的类方法并没有什么区别。 实际上，我们也确实可以通过 ClassTest.main(..)来调用 ClassTest 中的 main 方法。在 class 文件常量池字节码中有两个比较特别的方法名符号：<clinit>和<init>。其中<clinit>方法是编译器自己生成的，编译器会把类静态变量直接初始化语句和静态初始化语句块的代码都放到了 class 文件的<clinit>方法中。而对所有非静态非常量数据域的初始化工作要靠<init>方法来完成。针对每一个类的构造方法，编译器都会产生一个<init>方法。即使是缺省构造器也不例外。

为了说明 class 文件的结构，再看下面的 Java 代码：

```
public interface MyInterface {
    void hello();
}
```

编译上述 Java 代码后的字节码如图 6-2 所示。



图 6-2　编译后的字节码

在上述图 6-2 中，常量池数“0009”表示后面紧接着有 8 个常量池项，下面是 8 个常量池项的具体说明：

- 常量池索引 1 表示一个 CONSTANT_Class_info 表(07)，它引用索引为 7 的常量池；
- 常量池索引 2 表示一个 CONSTANT_Class_info 表(07)，它引用索引为 8 的常量池；
- 常量池索引 3 表示一个 CONSTANT_Utf8_info 表(01)，其内容为 hello(方法名)；
- 常量池索引 4 表示一个 CONSTANT_Utf8_info 表(01)，其内容为()V(方法参数与返回值)；
- 常量池索引 5 表示一个 CONSTANT_Utf8_info 表(01)，其内容为 SourceFile(某属性值)；
- 常量池索引 6 表示一个 CONSTANT Utf8 info 表(01)，其内容为 MyInterface.java(某属性值)；
- 常量池索引 7 表示一个 CONSTANT Utf8 info 表(01)，其内容为 MyInterface，被常量池索引 1 引用到(当前类)；
- 常量池索引 8 表示一个 CONSTANT Utf8 info 表(01)，其内容为 java/lang/Object，被常量池索引 2 引用到(超类)；
- 访问标志“0601”表示是 public(0001)、abstract(0400)，且是接口(0200)；

- 当前类索引号“0001”表示指向常量池索引 1，指明当前类为 MyInterface;
- 超类索引号“0002”表示指向常量池索引 2，指明超类为“java/lang/Object”；
- 实现的接口数“0000”表示没有实现任何接口;
- 字段数“0000”表示该接口没有字段;
- 方法数“0001”表示接口有一个方法。

剩下的字节码是方法列表及属性列表。

6.4 特殊字符串

常量池中容纳的符号引用包括三种特殊的字符串：全限定名、简单名称和描述符。所有的符号引用都包括类或者接口的全限定名。字段的符号引用除了全限定类型名之外，还包括简单字段名和字段描述符。方法的符号引用除了全限定类型名之外，还包括简单方法名和方法描述符。

在符号引用中使用的特殊字符串也同样用来描述被 Class 文件定义的类或者接口。例如，定义过的类或者接口会有一个全限定名。对于每一个在类或者接口中声明的字段，常量池中都会有一个简单名称和字段描述符。对于每一个在类或者接口中声明的方法，常量池中都会有一个简单名称和方法描述符。

6.4.1 全限定名

当常量池入口指向类或者接口时，它们给出该类或者接口的全限定名。在 Class 文件中，全限定名中的点用斜线取代了。例如，在 Class 文件中，java.lang.Object 的全限定名表示为 java/lang/Object；在 Class 文件中，java.util.Hashtable 的全限定名表示为“java/util/Hashtable”。

6.4.2 简单名称

字段名和方法名以简单名称(非全限定名)形式出现在常量池入口中。例如，一个指向类 java.lang.Object 所属方法 StringtoString()的常量池入口有一个形如“toStirng”的方法名。一个指向类 java.lang.System 所属字段 java.io.PrintStream.out 的常量池入口有个形如“out”的字段名。

6.4.3 描述符

除了类(或接口)的全限定名和简单字段(或方法)名，指向字段和方法的符号引用还包括描述符字符串。字段的描述符给出了字段的类型；方法描述符给出了方法的返回值和方法参数的数量、类型以及顺序。

字段和方法的描述符由如下所示的上下文无关语法定义。该语法中非终结符号用斜体字标出，如 FieldType；终结符号使用等宽度字体标出，如 B 或 V；星号代表紧接在它前面的符号(中间没有空格)将会出现 0 次或者多次。

表 6-2 中列出了每个基本类型终结符的含义。V 终结符表示方法返回值为 void 类型。8 种基本类型终结符中的每一个、返回值描述符终结符 V、对象类型终结符 L 和；数组类型终结符 I，以及方法描述符汇总终结符(和)，都是 ASCII 字符(除了空字符 null 外，能够对应于 ASCII 字符的每一个 Unicode 字符在 UTF-8 格式中，都可以使用相对应的 ASCII 字符值来描述)。对象类型中的 Class-name 部分为全限定名。这里的全限定名与 Class 文件中的全限定名一样，都用斜线取代了点。

表 6-2　基本类型终结符

终 结 符	类 型
B	byte
C	char
D	double
F	float
I	int
J	long
S	short
Z	boolean

表 6-3 列出了一些字段描述符的例子。在此需要注意的是：实例方法的方法描述符并没有包含作为第一个参数被传给所有实例方法的隐藏 this 参数。但所有调用实例方法的 Java 虚拟机指令都会隐式传递 this 参数。this 引用永远不会传给方法，因为类方法不会被对象调用。

表 6-3　字段描述符举例

描 述 符	说 明
I	int i;
[Ljava/lang/Object;	java.lang.Object[] obj;
([BII)Ljava/lang/String;	String method(byte[] b, int i, int j)
ZILjava/lang/String;II()Z	boolean method(boolean b, int i, String s, int j, int k)

6.5　常　量　池

常量池是一个可变长度 cp info 表的有序序列。cp info 表中的 tag(标志)项是一个无符号的 byte 类型值，它表明了表的类型和格式 cp info 表 一共有 11 种类型。

6.5.1　OCNSTANT_Utf8_info 表

可变长度的 CONSTANT Utf8 info 表使用 一种 UTF-8 格式的变体来存储 一个常量字

符串。这种类型的表可以存储多种字符串，包括以下几项。

(1) 文字字符串，如String对象。

(2) 被定义的类和接口的全限定名。

(3) 被定义的类的超类(如果有的话)的全限定名。

(4) 被定义的类和接口的父接口的全限定名。

(5) 由类或者接口声明的任意字段的简单名称和描述符。

(6) 由类或者接口声明的任意方法的简单名称和描述符。

(7) 任何引用的类和接口的全限定名。

(8) 任何引用的字段的简单名称和描述符。

(9) 任何引用的方法的简单名称和描述符。

(10) 与属性相关的字符串。

在CONSTANT_Utf8_info表中存储了四种基本信息类型：文字字符串，被定义的类和接口描述，对其他类或接口的符号引用以及与属性相关的字符串。一些与属性相关的字符串如：属性名称、产生class文件的源文件名称、局部变量的名称以及描述符。

UTF-8编码格式允许字符串中的所有Unicode字符以两个字节的形式表示，而ASCII字符(空字符null除外)以一个字节的形式表示。表6-4列出了CONSTANT_Utf8_info的格式。

表6-4 CONSTANT_Utf8_info表的格式

类 型	名 称	数 量
U1	tag	1
U2	length	1
Ul	bytes	length

CONSTANT_Utf8_info表中各项的具体说明如下。

- tag：tag项的值为CONSTANT_Utf8(1)。
- length：length项给出了后续bytes项的长度(字节数)。
- bytes：bytes项中包含按照变体UTF-8格式存储的字符串中的字符。从'\u0001'到'\u007f'的所有字符(除空字符null外所有的ASCII字符)都使用一个字节表示。

空字符null('\u0000')和从'\u0080'到'\u07ff'的所有字符使用两个字节表示从'\u0800'到'\uffff'的所有字符使用三个字节表示。

在OCNSTANT_Utf8_info表内，bytes项中的UTF-8字符串编码与标准UTF-8格式的区别在于：第一，在标准UTF-8编码模式中，空字符null使用一个字节表示；在CONSTANT_Utf8_info表中，空字符使用两个字节表示。这种对于空字符null的双字节编码，意味着bytes项的值永远不会为0；第二，bytes项中只使用了标准UTF-8编码中的单字节、双字节和三字节编码，而标准UTF-8编码还包括未在CONSTANT_Utf8_info表中使用的较长的格式。

6.5.2　CONSTANT_Integer_info 表

固定长度的 CONSTANT_Integer_info 表用来存储常量 int 类型值，该表只存储 int 值，不存储符号引用，表 6-5 列出了 CONSTANT_Integer_info 表的格式。

表 6-5　CONSTANT_Integer_info 表的格式

类　型	名　称	数　量
u1	tag	1
u4	bytes	1

CONSTANT_Integer_info 表中各项的具体说明如下：

- tag：tag 项的值为 CONSTANT_Integer(3)；
- bytes：bytes 项中按照高位在前的格式存储 int 类型值。

6.5.3　CONSTANT_Float_info 表

固定长度的 CONSTANT_Float_info 表用来存储常量 float 类型值，该表只存储 float 类型值，不存储符号引用，表 6-6 中列出了 CONSTANT_Float_info 表的格式。

表 6-6　CONSTANT_Float_info 表的格式

类　型	名　称	数　量
u1	tag	1
u4	bytes	1

CONSTANT_Float_info 表中各项的具体说明如下：

- tag：tag 项的值为 CONSTANT_Float(4)；
- bytes：bytes 项中按照高位在前的格式存储 float 类型值。

6.5.4　CONSTANT_Long_info 表

固定长度的 CONSTANT_Long_info 表用于存储 long 类型常量。该表中只存储 long 类型值，不存储符号引用。表 6-7 列出了 CONSTANG_Long_info 表的格式。

表 6-7　CONSTANT_Long_info 表的格式

类　型	名　称	数　量
u1	tag	1
u48	bytes	1

一个 long 类型值在常量池中占据常量池中的两个位置。在 class 文件中，一个 long 类型入口紧接着下一个入口，但下一个入口的索引值却比紧挨着的上一个入口的值多 2。

CONSTANT_Long_info 表中各项的具体说明如下：

- tag：Ttag 项的值为 CONSTANT_Long(5)；
- bytes：bytes 项中按照高位在前的格式存储 long 类型值。

6.5.5 CONSTANT_Double_info 表

固定长度的 CONSTANT_Double_info 表用来存储 double 类型常量。该表只用来存储 double 值，不存储符号引用。表 6-8 列出了 CONSTANT_Double_info 表的格式。

表 6-8 CONSTANT_Double_info 表的格式

类 型	名 称	数 量
u1	tag	1
u48	bytes	1

一个 double 类型值在常量池中占据常量池中的两个位置。在 class 文件中，double 入口的下一个入口紧随其后，但下一入口的索引值却要比上一个入口的索引值大 2。

CONSTANT_Double_info 表中各项的具体说明如下：

- tag：tag 项的值为 CONSTANT_Double(6)；
- bytes：bytes 项中按照高位在前的格式存储 double 类型值。

6.5.6 CONSTANT_Class_info 表

固定长度的 CONSTANT_Class_info 表使用符号引用来描述类或者接口。无论指向类、接口、字段还是方法，所有的符号引用都包含一个 CONSTANT_Class_info 表。表 6-9 列出了 CONSTANT_Class_info 表的格式。

表 6-9 CONSTANT_Class_info 表的格式

类 型	名 称	数 量
u1	tag	1
u48	Name_index	1

CONSTANT_Class_info 表中各项的具体说明如下：

- tag：tag 项的值为 CONSTANT_Class(7)；
- name_index：name_index 项给出了包含类或者接口全限定名 CONSTANT_Utf8_info 表的索引。

由于 Java 中的数组是完善的对象，CONSTANT_Class_info 表也能够用来描述数组类。CONSTANT_Class_info 表中的 name_index 项指向 CONSTANT_Utf8_info 表，该表中包含了数组的描述符，描述符可作为数组类的名称。例如，一个 double[][]数组类型的类名为它的描述符 [[D;net.jini.core.llokup.ServiceItem[][] 数组类型的类名为它的描述符 [[[Lnet/jinni/core/lookup/ServiceItem;。由于 Java 数组最多只能有 255 维，数组描述符最多只能有 255 个引导符“[”。

6.5.7　CONSTANT_String_info 表

固定长度的 CONSTANT String_info 表用来存储文字字符串值，该值亦可标识为类 java.lang.String 的实例。该表中只存储文字字符串值，不存储符号引用。表 6-10 列出了 CONSTANT String info 表的格式。

表 6-10　CONSTANT_String_info 表的格式

类　型	名　称	数　量
u1	Tag	1
u48	string_index	1

CONSTANT_String_info 表中各项的具体说明如下：

- tag：tag 项的值为 CONSTANT_String(8)；
- string_index：String_index 项给出了包含文字字符串的 CONSTANT_Utf8_info 表的索引。

6.5.8　CONSTANT_Fieldref_info 表

固定长度的 CONSTANT_Fieldref_info 表描述了指向字段的符号引用。表 6-11 列出了 CONSTANT_Fieldref_info 表的格式。

表 6-11　CONSTANT_Fieldref_info 表的格式

类　型	名　称	数　量
u1	tag	1
u2	class_index	1

CONSTANT_Fieldref_info 表中各项的具体说明如下：

- Tag：Tag 项的值为 CONSTANT_Fieldref(9)；
- Class_index：Class_index 项给出了声明被引用字段的类或者接口的 CONSTANT_Class_info 入口的索引。

需要注意的是，由 class_index 指定的 CONSTANT_Class_info 不只是代表类，还可能代表接口。尽管接口中能够声明字段，而且可以分别声明为公开、静态和 final 类型。但如前所述，如果其他类的静态 final 字段使用编译时的常量进行初始化操作，那么 Class 文件不包含对这些字段的符号引用。但是，Class 文件可以包含它使用的任何这些静态 final 字段的常量值的复本。例如，如果类使用在接口中声明的 float 类型的静态 final 字段，而且它被初始化为编译时的常量，该类将会在它自己的存储 float 值的常量池中拥有一个 CONSTANT Float info 表。但是如果该接口使用只有在运行时才能计算出的表达式来初始化它的静态 final 字段，那么在使用该字段的类的常量池中，将会有一个对该接口中的字段进行符号引用的 CONSTANT Fieldref info 表。

- Name and type index：Name and type index 提供了 CONSTANT NameAndType

info 入口的索引，该入口提供了字段的简单名称以及描述符。

6.5.9 CONSTANT_Methodref_info 表

固定长度的 CONSTANT_Methodref_info 表使用符号引用来描述类中声明的方法(不包括接口中的方法)。表 6-12 列出了 CONSTANT_Methodref_info 表的格式。

表 6-12 CONSTANT_Methodref_info 表的格式

类 型	名 称	数 量
u1	tag	1
u2	class_index	1
u2	Name_and_type_index	1

CONSTANT_Methodref_info 表中各项的具体说明如下：

- Tag：Tag 项的值为 CONSTANT_Methodref(10)；
- Class_index：Class_index 项给出了声明被引用方法的 CONSTANT_Class_info 入口的索引。由 class_inex 所指定的 CONSTANT_Class_info 表必须为类，不能为接口。指向接口中声明的方法的符号引用使用 CONSTANT_InterfaceMethodref 表；
- Name_and_type_index：提供了 CONSTANT_NameAndType_info 入口的索引，该入口提供了方法的简单名称以及描述符。如果方法的简单名称开始与“<”)('\u003c')符号，该方法必须为一个实例初始化方法。它的简单名称必须为<init>，它的返回值必须为 void 类型。否则，该方法的名称必须为一个有效果的 Java 程序设计语言的标识符。

6.5.10 CONSTANT_InterfaceMethodref_info 表

固定长度的 CONSTANT_InterfaceMethodref_info 表使用符号引用来描述接口中声明的方法(不包括类中的方法)。表 6-13 列出了 CONSTANT_InterfaceMethodref_info 表的格式。

表 6-13 CONSTANT_InterfaceMethodref_info 表的格式

类 型	名 称	数 量
u1	tag	1
u2	class_index	1
u2	Name_and_type_index	1

表 CONSTANT_InterfaceMethodref_info 中各项的具体说明如下：

- tag：Tag 项的值为 CONSTANT_InterfaceMethodref(11)；
- class_index：此项给出了声明了被引用方法接口 CONSTANT_Class_info 的入口索引。由 class_index 所指定的 CONSTANT_Class_info 表必须为接口，不能为类。在类中生命的方法的符号引用使用 CONSTANT_Methodref 表；
- name_and_type_index：提供了 CONSTANT_NameAndType_info 入口的索引，该

入口提供了方法的简单名称以及描述符。

6.5.11　CONSTANT_NameAndType_info 表

固定长度的 CONSTANT_NameAndType_info 表构成了指向字段或者方法的符号引用的一部分。该表提供了所引用字段或者方法的简单名称和描述符的常量池入口。表 6-14 列出了 CONSTANT_NameAndType_info 表的格式。

表 6-14　CONSTANT_NameAndType_info 表的格式

类　型	名　称	数　量
u1	tag	1
u2	name_index	1
u2	descriptor_index	1

CONSTANT_NameAndType_info 表中各项的具体说明如下：

- tag：Tag 项的值为 CONSTANT_NameAndType(12)；
- name_index：此项给出了 CONSTANT_Utf8_info 入口的索引，该入口给出了字段或者方法的名称。该名称可以是一个有效的 Java 程序设计语言的标识符，也可以是<init>；
- descriptor_index：提供了 CONSTANT_Utf8_info 入口的索引，该入口提供了字段或者方法的描述符。该描述符必须为一个有效的字段或者方法的描述符。

6.6　字　　段

在类或者接口中声明的每一个字段(类变量或者实例变量)，都由 Class 文件中的一个名为 field_info 的可变长度的表进行描述。在一个 Class 文件中，不会存在两个具有相同名字和描述符的字段。需要注意的是，尽管在 Java 程序设计语言中不会有两个相同名字的字段存在于同一个类或者接口中，但一个 Class 文件中的两个字段可以拥有同一个名字——只要它们的描述符不同。换句话说，尽管在程序设计语言中，无法在同一个类或者接口中定义两个具有同样名字和不同类别的字段，但是两个这样的字段却可以同时合法地出现在一个 Java Class 文件中。field_info 表中各项的具体说明如下。

(1) Access_flags：声明字段时使用的修饰符存放在字段的 access_flags 项中。

类(不包括接口)中声明的字段，只能拥有 ACC_PUBLIC、ACC_PRIVATE、ACC_PROTECTED 这三个标志中的一个，不能同时设置 ACC_FINAL 和 ACC_VOLATILE。所有接口中声明的字段必须有且只能有 ACC_PUBLIC、ACC_STATIC 和 ACC_FINAL 这三种标志。

Access_flags 中没有用到的位都被设为 0，Java 虚拟机实现将忽略它们。

(2) Name_index：提供了给出字段简单名称(不是全限定名)的 CONSTANT_Utf8_info 入口的索引。在 Class 文件中的每一个字段的名称都必须符合 Java 程序设计语言中对名称

的有效规定。

(3) Descriptor_index：提供了给出字段描述符的 CONSTANT_Utf8_info 入口的索引。

(4) Arrtibutes_count 和 attributes：attributes 项是由多个 attribute_info 表组成的列表。Attributes_count 指出了列表中 attribute_info 表的数量。一个字段在其列表中可以有任意数量的属性。由 Java 虚拟机规范定义的三种可能会出现在此项中的属性是：ConstantValue，Deprecated 和 Synthetic。Java 虚拟机唯一需要识别的属性是 ConstantValue 属性。虚拟机实现必须忽略任何无法识别的属性。

6.7 方　　法

在 Class 文件中，每个在类和接口中声明的方法，或者由编译器产生的方法，都由一个可变长度的 method_info 表来描述。同一个类中不能存在两个名字及描述符完全相同的方法。需要注意的是，在 Java 程序设计语言中，尽管在同一个类或者接口中声明的两个方法不能有同样的特征签名(除返回类型之外的描述符)，但是在同一个 Class 文件中，两个方法可以拥有同样特征的签名，前提是它们的返回值不能相同。换句话说，在 Java 源文件的同一个类里，如果声明了两个具有相同名字和相同参数类型，但返回值不同的方法，这个程序将无法编译通过。在 Java 程序设计语言中，不能仅仅通过返回值的不同来重载方法。但是同样的两个方法可以和谐地在一个 Class 文件中共存。

有可能在 Class 文件中出现的两种编译器产生的方法是：实例初始化(名为<init>)和类与接口初始化方法(名为<clinit>)。method_info 表中各项的具体说明如下。

(1) Access_flags：在声明方法时使用的修饰符存放在方法的 access_flags 项中。

类(不包括接口)中声明的方法只能拥有 ACC_PUBLIC、ACC_PROVATE、ACC_PROTECTED 这三个标志中的一个。如果设定了一个方法的 ACC_ABSTRACT 标志，那么它的 ACC_PRIVATE、ACC_STATIC、ACC_FINAL、ACC_SYNCHRONIZED、ACC_NATIVE 以及 ACC_STRICT 标志都必须清除。接口中声明的所有方法必须有 ACC_PUBLIC 和 ACC_ABSTRACT 标志。除此以外，接口方法不能使用其他标志，但接口初始化方法(<clinit>)可以使用 ACC_STRICT 标志。

实例初始化方法(<init>)可以只使用 ACC_PUBLIC、ACC_PRIVATE 和 ACC_PROTECTED 标志。因为类与接口初始化方法(<clinit>)只由 Java 虚拟机直接调用，永远不会被 Java 字节码直接调用，这样，<clinit>方法的 access_flags 中的标志位，除去 ACC_STRICT 之外的所有位都应该被忽略。

在 access_flags 中没有用到的位都被设为 0，Java 虚拟机实现也将忽略它们。

(2) Name_index：提供了 CONSTANT_Utf8_info 入口的索引，该入口给出了方法的简单名称(不是全限定名)。在 Class 文件中的每一个方法的名称，都必须或者为<init>，或者为<clinit>，或者是 Java 程序设计语言中有效的方法名称(简单名称，不是全限定名)。

(3) descriptor_index：提供了 CONSTANT_Utf8_info 入口的索引，该入口给出了方法的描述符。

(4) attributes 和 Attributes count：attributes 项是由多个 attribute_info 表组成的列表。Attribute_count 给出了列表中 attribute_info 表的数量。一个字段在其列表中可以有任意数量的属性。在此项中可能会出现的由 Java 虚拟机规范定义的 4 种属性是 Code、Deprecated、Exceptions 和 Synthetic。这 4 种属性将在本章后面进一步阐述。Java 虚拟机只需要识别 Code 和 Exception 属性。虚拟机实现必须忽略任何无法识别的属性。

6.8　属　　性

Java class 文件可以出现在 ClassFile、field_info、method_info 和 Code_attribute 表中。Code_attribute 表本身即为一个属性，在 Java 虚拟机规范定义了 9 种属性。为了正确地解释 Java Class 文件，所有 Java 虚拟机实现都必须能够识别下列三种属性，分别是 Code、ConstantValue 和 Exception。为了正确地实现 Java 和 Java 2 平台类库，虚拟机实现必须能够识别 InnerClasses 和 Synthetic 属性，但可以自主选择究竟是识别还是忽略其他一些与定义的属性。

如果需要向 Java class 文件中加入新的属性，除 Sun 公司以外的任何人都必须遵循下列两个步骤：

(1) 任何不是由规范进行与定义的属性都不能影响类或者接口类型的语义。新的属性只能向 Class 文件添加新的信息，如在调试过程中用到的信息等。

(2) 属性必须使用与 Internet 域名方案颠倒的命名方式，Internet 域名方案是针对 Java 语言规范中包的命名所定义的。例如，如果拥有一个 Internet 域名 aaa.com，而需要创建的新属性为 CompilerVersion，那么属性则应该命名为 com.aaa.CompilerVersion。

6.8.1　属性格式

属性 Attribute_name_index 的前两个字节构成了到 CONSTANT_Utf8_info 表的常量池的索引，表 CONSTANT_Utf8_info 中包含了属性的字符串名称。因此，每一个 attribute_info 表，由其表中的第一项指出了它的“类型”。

紧随 attribute_name_index，后面是 4 字节长的 attribute_length 项，它给出除去起始 6 个字节后整个 attribute_info 表的长度(attribute_length 项可以为 0)。因为只要遵循一定规则(如下所列)，任何人都能够向 Java Class 文件中加入属性，所以长度是不可缺少的。Java 虚拟机实现能够识别新属性，实现必须忽略任何无法识别的属性。当解析 Class 文件时，attribute_length 允许虚拟机跳过无法识别的属性。

表 Attribute_info 中各项的具体说明如下：

- Attribute name index 项：给出了包含属性名称的 CONSTANT Utf8 入口的常量池的索引。
- Attribute_lengthattribute_length 项：给出了属性数据的长度(以字节计)，但是包括 attribute_name_index 和 attribute_length 在内的起始 6 个字节不包括在内。
- Info 项：包含属性数据。

6.8.2 Code 属性

在可变长度的 Code attribute 表中，定义了方法的字节码序列和其他信息。在所有不是抽象或者本地方法的 method info 信息中，都存在一个 Code_attribute 表。表 Code attribute 中各项的具体说明如下。

- Attribute_name index 项：给出包含字符串“Code”的 CONSTANT Utf8_info 入口的常量池索引。
- Attribute_length 项：给出除去起始 6 个字节(包含 attribute_name_index 和 attribute_length 项)后，Code 属性以字节为单位的长度。
- Max_stack：在方法执行中的任意时刻，max_stack 项给出该方法的操作数栈的最大长度(以字节为单位)。
- Max_locals 项：给出方法的局部变量所需存储空间的长度(以字为单位)。无论虚拟机什么时候调用被 Code 属性所描述的方法，它都必须分配一个长度为 max_locals 的局部变量数组。这个数组用来存储传递给方法的参数以及为方法所使用的局部变量。Long 或者 double 类型值的最大有效的局部变量索引是 max_locals=2.任何其他类型值的最大有效局部变量索引为 max_locals=1.
- Code_length 和 code：Code_length 项给出该方法字节码流的长度(按字节计)。字节码本身将会出现在 code 项中。Code_length 的值必须大于 0。
- Exception_table_length 和 exception_table：Exception_table_length 项是一个 exception_info 表的列表。每个 exception_info 表都描述了一个异常表项。Exception_table_length 项给出了 exception_table 中 exception_info 表的数目。Exception_table 表在列表中按照方法执行抛出异常时 Java 虚拟机检查匹配异常处理器(catch 子句)的顺序进行排列。
- Attributes_count 和 attributes：Attributes 项是一个 attribute_info 表的列表。Attributes_count 项给出了列表中 attribute_info 表的数目。该项中可以出现 Java 虚拟机规范所定义的两种属性：LineNumberTable 和 LocalVariableTable。Java 虚拟机实现允许忽略 Code 属性内 attributes 项中的任何属性，如果 Java 虚拟机无法识别这些，Java 虚拟机必须忽略它们。

固定长度的 exception_info 表描述了一个异常表项，该表在 Code 属性中的 exception info 项中出现，它是 exception_info 表序列的组成部分。表 Exception info 中各项的具体说明如下。

- Start pc 项：给出从代码数组起始处到异常处理器起始处的偏移量。
- End pc 项：给出从代码数组起始处到异常处理器结束后一个字节的偏移量。
- Handler pc 项：如果抛出的异常被该项捕获的话，则给出一条从代码数组起始处跳转到异常处理器的第一条指令偏移量的指令。
- Catch type 项：给出被该异常处理器所捕获的异常类型的 CONSTANT Class info 入口的常量池索引。CONSTANT Class info 入口必须描述了类 java.lang.Throwable 或其子类。

如果 catch type 的值为 0(不是一个有效的常量池索引，因为常量池从索引 1 开始)，那么异常处理器将处理所有的异常。一个值为 0 的 catch_type 用于事项 finally 子句。

6.8.3 ConstantValue 属性

固定长度的 ConstantValue 属性出现在值为常量的字段的 field info 表中。在给定 field info 表的属性项中，最多只可能出现一个 ConstantValue 属性。在包含 ConstantValue 属性的 field_info 表内的 access_flag 中，必须设定 ACC STATIC 标志。尽管可能并不需要，但是也可以设定 ACC_FINAL 标志。当虚拟机初始化一个具有 ConstantValue 属性的字段时，它将一个常量值赋给这个字段。赋值操作的虚拟机调用声明此字段的类或者接口的初始化方法之前进行。

表 ConstantValue_attribute 中各项的具体说明如下。

- Attribute_name_index 项：给出包含字符串“ConstantValue”的 CONSTANT_Utf8_info 入口的常量池索引。
- Attribute_length 项：ConstantValue_attribute 中的 attribute_length 项的值永远为 2。
- Constantvalue_index 项：给出提供常量值的入口的常量池索引。

6.8.4 Deprecated 属性

固定长度的 Deprecated 属性存在于 field_info、method_info 和 ClassFile 表内的 attributes 项中，这是一个可选的项，它指出了所禁用的字段、方法或者类型。这里禁用的意思是尽管一个字段、方法或者类型仍然存在于执行的方法中，但程序员永远再不会用到它。更确切地说，程序员使用其他相近的字段、方法和类型，而不会使用所禁用的项。

对于编译器、虚拟机或者用来读取 class 文件的任何工具来说，都可以使用属性 Deprecated 来通知程序员程序使用了禁用的字段、方法或者类型。从 JDK 1.1 版本开始便引入了 Deprecated 属性，用来支持 javadoc 工具使用的文档注释中的@deprecated 标志。

6.8.5 Exception 属性

可变程度的 Exception 属性列出了方法可能抛出的异常。Exception_attribute 表会出现在每一个可能抛出已检出异常的方法的 method_info 表中。

如果一个方法是 RuntimeException、Error 或者是在方法的 Exceptions 属性中所列出的异常的实例或者子类，那么它应该只抛出一个异常。尽管这条规则应该是被 Java 编译器强制执行，但它没有被 Java 虚拟机强制执行。因此为了 Java 编译器，Exceptions 属性存在于 Java Class 文件中。表 Exceptions attribute 中各项的具体说明如下。

- Attribute name index：给出包含字符串“Exceptions”的 CONSTANT Utf8 info 入口的常量池索引。
- Attribute_length：给出了除去起始 6 个字节(其中包含 Attribute_name_index 和 Attribute_length 项)后，Exception_attribute 的长度(按字节计算)。
- Number_of_exceptioins 和 Exception_index_table：Exception_index_table 是一个该

方法内 throws 子句所声明异常的常量池中 CONSTANT_Class_info 入口的索引的数组。换句话说，Exception_index_table 列出了该方法可能抛出的所有已检出的异常。Number_of_exceptioins 项指出了数组中索引的数目。

6.8.6 InnerClasses 属性

可变长度的 InnerClasses 属性对名字、访问标志以及被声明为成员的任何嵌入类型的外围类型，或者用别的方法由类或者接口陈述的类型(嵌入类型不是包中的成员，而是类或者接口的成员)。如果类或者接口的代码指向一个嵌入类型，该类或者接口的常量池将会包含一个用于此嵌入类型的 CONSTANT_Class_info 入口，这些嵌入类型被定义为类或者数组的直接成员——尽管类或者接口并没有另外描述内嵌类型。如果类或者接口的常量池包含让你和内嵌类型的 CONSTANT_Class_info 入口，那么此类或者接口的 Class 文件必须包含一个 InnerClasses_attribute 表。此表存在于它自身的 ClassFile 表的 attributes 项中。

6.9 JVM 加载 Class 文件的原理

Java 中的所有类，必须被装载到 JVM 中才能运行，这个装载工作是由 JVM 中的类装载器完成的，类装载器所做的工作实质是把类文件从硬盘读取到内存中。本节将简要讲解 JVM 加载 Class 文件的基本原理。

6.9.1 Java 中的类文件

Java 中的类可以大致分为以下三种：

- 系统类。
- 扩展类。
- 由程序员自定义的类。

类的装载方式有以下两种。

- 隐式装载：程序在运行过程中当碰到通过 new 等方式生成对象时，隐式调用类装载器加载对应的类到 JVM 中。
- 显式装载：通过 class.forname()等方法，显式加载需要的类。

一个应用程序总是由 n 个类组成，当 Java 程序启动时，并不是一次把所有的类全部加载后再运行，它总是先把保证程序运行的基础类一次性加载到 JVM 中，其他类等到 JVM 用到的时候再加载，这样的好处是节省了内存的开销，因为 Java 最早就是为嵌入式系统而设计的，这是一种可以理解的机制，而用到时再加载这也是 Java 动态性的一种体现。

Java 中的类装载器实质上也是类，功能是把类载入 JVM 中，值得注意的是 JVM 的类装载器是三个，而不是一个，具体层次结构如下：

```
Bootstrap Loader  - 负责加载系统类
      |
    - - ExtClassLoader  - 负责加载扩展类
```



使用三个类加载器的原因有两点：一方面是分工，各自负责各自的区块；另一方面为了实现委托模型，下面会谈到该模型。

因为在 Java 中有三个类加载器，所以当碰到一个类需要加载时，需要面对如何协调工作的问题，即 Java 是如何区分一个类该由哪个类加载器来完成的问题。在这里 Java 采用了委托模型机制，这个机制简单来讲，就是"类装载器有载入类的需求时，会先请示其 Parent 使用其搜索路径帮忙载入，如果 Parent 找不到，那么才由自己依照自己的搜索路径搜索类"。

下面举一个例子来说明，为了更好地理解，先弄清楚几行代码：

```
Public class Test{
    Public static void main(String[] arg){
      ClassLoader c  = Test.class.getClassLoader();  //获取 Test 类的类加载器
        System.out.println(c);
      ClassLoader c1 = c.getParent();  //获取 c 这个类加载器的父类加载器
        System.out.println(c1);
      ClassLoader c2 = c1.getParent();//获取 c1 这个类加载器的父类加载器
        System.out.println(c2);
  }
}
```

运行后会输出：

```
    。。。AppClassLoader
    。。。ExtClassLoader
    Null
```

由此可以看出，Test 是由 AppClassLoader 加载器加载的。AppClassLoader 的 Parent 加载器是 ExtClassLoader。Bootstrap Loader 是用 C++语言写的，根据 Java 的观点来看，逻辑上并不存在 Bootstrap Loader 的类实体，所以在 Java 程序代码里试图打印出其内容时会看到输出为 null。

类装载器 ClassLoader(一个抽象类)用于描述 JVM 加载 Class 文件的原理机制。类装载器就是寻找类或接口字节码文件进行解析并构造 JVM 内部对象表示的组件，在 Java 类装载器中把一个类装入 JVM 的具体步骤如下：

(1) 装载：查找和导入 Class 文件。

(2) 链接：其中解析步骤是可以选择的。

① 检查：检查载入的 Class 文件数据的正确性。

② 准备：给类的静态变量分配存储空间。

③ 解析：将符号引用转成直接引用。

(3) 初始化：对静态变量，静态代码块执行初始化工作。

类装载工作由 ClassLoder 和其子类负责，在运行 JVM 时会产生三个 ClassLoader。

- 根装载器：不是 ClassLoader 的子类，由 C++编写，因此在 java 中看不到他，负责装载 JRE 的核心类库，如 JRE 目录下的 rt.jar、charsets.jar 等。
- ExtClassLoader：扩展类装载器，是 ClassLoder 的子类，负责装载 JRE 扩展目录

ext 下的 jar 类包。

- AppClassLoader：负责装载 classpath 路径下的类包。

上述三个类装载器存在父子层级关系，即根装载器是 ExtClassLoader 的父装载器，ExtClassLoader 是 AppClassLoader 的父装载器。在默认情况下，使用 AppClassLoader 装载应用程序的类。

Java 装载类使用“全盘负责委托机制”。“全盘负责”是指当一个 ClassLoder 装载一个类时，除非显示的是使用另外一个 ClassLoder，该类所依赖及引用的类也由这个 ClassLoder 载入的。“委托机制”是指先委托父类装载器寻找目标类，只有在找不到的情况下才从自己的类路径中查找并装载目标类。这一点是从安全方面考虑的，试想如果一个人写了一个恶意的基础类(如 java.lang.String)并加载到 JVM 中，这将将会引起严重的后果。但有了“全盘负责制”之后，java.lang.String 永远是由根装载器来装载，避免了以上情况的发生

除了 JVM 默认的三个 ClassLoder 以外，第三方可以编写自己的类装载器，以实现一些特殊的需求。当类文件被装载解析后，在 JVM 中都有一个对应的 java.lang.Class 对象，提供了类结构信息的描述。数组、枚举、基本数据类型甚至 void 都拥有对应的 Class 对象。Class 类没有 public 的构造方法，Class 对象是在装载类时由 JVM 通过调用类装载器中的 defineClass()方法自动构造的。

ClassLoader 中的主要方法如下：

```
public Class<?> loadClass(String name)
                throws ClassNotFoundException
```

参数“name”用于指定类装载器需要装载类的名字，必须使用全限定类名。该方法有一个重载方法 loadClass(String name,Boolean resolve)，resolve 参数告诉类装载器是否解析该类。在初始化类之前应考虑进行类解析的工作，但并不是所有类都需要解析，如果 JVM 只需要知道该类是否存在或找出该类的超类，那么就不需要进行解析。

6.9.2 JVM 加载 Class 文件

当我们使用命令来执行某一个 Java 程序时，例如执行 Test.class 的时候，具体过程如下。

(1) java.exe 会帮助我们找到 JRE，接着找到位于 JRE 内部的 jvm.dll ，这才是真正的 Java 虚拟机器，最后加载动态库以激活 Java 虚拟机器。

(2) 当激活虚拟机器以后，会先做一些初始化的动作，比如说读取系统参数等。一旦初始化动作完成之后，就会产生第一个类装载器——Bootstrap Loader(启动类装载器)。

(3) Bootstrap Loader 所做的初始工作中，除了一些基本的初始化动作之外，最重要的就是加载 Launcher.java 之中的 ExtClassLoader(扩展类装载器)，并设定其 Parent 为 null，代表其父加载器为 BootstrapLoader。

(4) 然后 Bootstrap Loader 再要求加载 Launcher.java 之中的 AppClassLoader(用户自定义类装载器)，并设定其 Parent 为之前产生的 ExtClassLoader 实体。这两个加载器都是以静态类的形式存在的。

在此需要注意的是，Launcher$ExtClassLoader.class 与 Launcher$AppClassLoader.class 都是由 Bootstrap Loader 所加载，所以 Parent 和由哪个类加载器加载没有关系。

1．类装载器体系结构

在 JVM 加载 class 文件时必须通过一个叫做类装载器的程序，它的作用就是从磁盘文件中将要运行代码的字节码流加载进内存(JVM 管理的方法区)中。下面是几个比较重要的概念。

1) 启动类装载器

每个 Java 虚拟机实现都必须有一个启动类装载器。它只负责在系统类(核心 Java API 的 Class 文件)的安装路径中查找要装入的类。这个装载器的实现由 C++ 所撰写而成，是 JVM 实现的一部分。

2) 扩展类装载器和自定义类装载器

负责除核心 Java API 以外的其他 Class 文件的装载。例如用于安装或下载标准扩展的 class 文件，在类路径中发现的类库的 Class 文件，用于应用程序运行的 Class 文件等。这里有一点需要注意，自定义类装载器并非由应用程序员自己实现，它也是 JVM。

3) 命名空间

Java 虚拟机为每一个类装载器维护一个唯一标识的命名空间。一个 Java 程序可以多次装载具有同一个全限定名的多个类。Java 虚拟机要确定这“多个类”的唯一性，因此当多个类装载器都装载了同名的类时，为了唯一地标识这个类，还要在类名前加上装载该类的类装载器的标识，指出了类所位于的命名空间。

命名空间有助于安全的实现，因为可以有效地在装入了不同命名空间的类之间设置一个防护罩。在 Java 虚拟机中，在同一个命名空间内的类可以直接进行交互，而不同的命名空间中的类甚至不能察觉彼此的存在，除非显式地提供了允许它们进行交互的机制。一旦加载后，如果一个恶意的类被赋予权限访问其他虚拟机加载的当前类，它就可以潜在地知道一些它不应该知道的信息，或者干扰程序的正常运行。

2．双亲委托模型

用户自定义类装载器经常依赖其他类装载器，至少依赖于虚拟机启动时创建的启动类装载器来帮助它实现一些类装载请求。在 JDK1.2 版本前，非启动类装载器必须显式地求助于其他类装载器，类装载器可以请求另一个用户自定义的类装载器来装载一个类，这个请求是通过对被请求的用户自定义类装载器调用 loadClass()来实现的。除此以外，类装载器也可以通过调用 findSystemClass()来请求启动类装载器来装载类，这是类 ClassLoader 中的一个静态方法。

在 JDK1.2 及其以后的版本中，类装载器请求另一个类装载器来装载类型的过程被形式化，这被称为双亲委派模式。从 JDK1.2 版本开始，除启动类装载器以外的每一个类装载器外，都有一个“双亲”类装载器，在某个特定的类装载器试图以常用方式装载类型以前，它会先默认地将这个任务“委派”给它的双亲——请求它的双亲来装载这个类型。这个双亲再依次请求它自己的双亲来装载这个类型。这个委派的过程一直向上继续，直到达到启动类装载器，通常启动类装载器是委派链中的最后一个类装载器。如果一个类装载器

的双亲类装载器有能力来装载这个类型，则这个类装载器返回这个类型。否则，这个类装载器试图自己来装载这个类。

当 Java 虚拟机开始运行时，在应用程序开始启动以前，它至少创建一个用户自定义装载器，也可能创建多个。所有这些装载器被连接在一个 Parent-Child 的委托链中，在这个链的顶端是启动类装载器。

Java

第 7 章

栈和局部变量操作

Java 虚拟机通过装载、连接和初始化一个 Java 类型，使该类型可以被正在运行的 Java 程序所使用。本章将详细讲解 Java 虚拟机实现栈和局部变量操作的基本知识，为读者学习本书后面的知识打下基础。

7.1 类型装载、连接和初始化

Java 虚拟机通过装载、连接和初始化一个 Java 类型，使该类型可以被正在运行的 Java 程序所使用。具体功能如下。

- 装载：功能是把二进制形式的 Java 类型读入 Java 虚拟机中，
- 连接：功能是把这种已经读入虚拟机的二进制形式的数据合并到虚拟机运行时的状态中去。连接阶段分为 3 个子步骤：验证、准备和解析。“验证”步骤确保了 Java 类型数据格式正确并且适于 Java 虚拟机使用；而“准备”步骤则负责为该类型分配它所需的内存，比如，为它的类变量分配内存；“解析”步骤则负责把常量池中的符号引用转换为直接引用。虚拟机的实现可以推迟解析这一步，它可以在当运行中的程序真正使用某个符号引用时再去解析它(把该符号引用转换为直接引用)。当验证准备和(可选的)解析步骤都完成了时，该类型就已经为初始化做好了准备。
- 初始化：在此期间，都将给类变量赋以适当的初始值。

上述装载、连接和初始化这三个阶段必须按顺序进行，唯一的例外就是连接阶段的第三步——解析，它可以在初始化之后再进行。

在类和接口被装载和连接的时机上，Java 虚拟机规范给具体实现提供了一定的灵活性。但是它严格定义了初始化的时机。所有的 Java 虚拟机实现必须在每个类或接口首次主动使用时初始化。下面这 6 种情形符合主动使用的要求。

(1) 当创建某个类的新实例时，或者通过在字节码中执行 new 指令，或者通过不明确的创建、反射、克隆或者反序列化时。

(2) 当调用某个类的静态方法时，即在字节码中执行 invokestatic 指令时。

(3) 当使用某个类或接口的静态字段，或者对该字段赋值时。即在字节码中，执行 getstatic 或 putstatic 指令时，用 final 修饰的静态字段除外，它被初始化为一个编译时的常量表达式。

(4) 当调用 Java API 的某些反射方法时，比如，类 Class 中的方法或者 java.lang.reflect 包中的类的方法。

(5) 当初始化某个类的子类时。即当初始化某个类时，要求它的超类已经被初始化了。

(6) 当虚拟机启动时某个被表明为启动类的类，即含有 main()方法的那个类。

除了上述这 6 种情形外，所有其他使用 Java 类型的方式都是被动使用的，他们都不会导致 Java 类型的初始化。

在上面我们曾提到，任何一个类的初始化都要求它的超类在此之前已经初始化了。以此类推，该规则就意味着某个类的所有祖先类必须在该类之前被初始化。然而，对于接口来说，这条规则并不适用。只有在某个接口所声明的非常量字段被使用时，该接口才会被初始化，而不会因为实现这个接口的子接口或类要初始化而被初始化。因而，任何一个类

的初始化都要求它的所有祖先类(而不是祖先接口)预先被初始化。而一个接口的初始化，并不要求它的祖先接口预先被初始化。

“在首次主动使用时初始化”这个规则直接影响着装载、连接和初始化类的机制。在首次主动使用时，其类型必须被初始化。然而，在类型能被初始化之前，它必须已经被连接了，而在它能被连接之前，它必须已经被加载了。Java 虚拟机的实现可以根据需要在更早的时候装载以及连接类型，没有必要一直要等到该类型的首次主动使用采取装载和连接它。无论如何，如果一个类型在它的首次主动使用之前还没有被装载和连接的话，那它必须在此时被装载和连接，这样它才能被初始化。

7.1.1　装载

装载阶段由三个基本动作组成，要装载一个类型，Java 虚拟机必须完成如下三个基本动作。

(1) 通过该类型的完全限定名，产生一个代表该类型的二进制数据流。

(2) 解析这个二进制数据流为方法区内的内部数据结构。

(3) 创建一个表示该类型的 java.lang.Class 类的实例。

这个二进制数据流可能遵守 java class 文件格式，但是也可能遵守其他的格式。就像前一章提到的那样，所有的 Java 虚拟机实现必须能识别 Java Class 文件格式，但是个别的实现也可以识别其他的二进制格式。

Java 虚拟机规范并没有说 Java 类型的二进制数据应该怎样产生。下面是一些可能的产生“类型的二进制数据”的方式。

(1) 从本地文件系统装载一个 Java Class 文件。

(2) 通过网络传输一个 Java Class 文件。

(3) 从一个 ZIP、Jar、CAB 或者其他某种归档文件中提取 Java Class 文件。

(4) 从一个专有数据库中提取 Java Class 文件。

(5) 把一个 Java 源文件动态编译为 Class 文件格式。

(6) 动态为某个类型计算其 Class 文件数据。

(7) 使用上述任何方法，但是使用不同于 Java Class 文件的其他二进制文件格式。

有了类型的二进制数据之后，Java 虚拟机必须对这些数据进行足够的处理，然后它才能创建 java.lang.Class 的实例对象。虚拟机必须把这些二进制数据解析为与实现相关的内部数据结构。装载步骤的最终产品就是这个 Class 类的实例对象，它成为 Java 程序与内部数据之间的接口。要访问关于该类型的信息(它们是存储在内部数据结构中的)，程序就要调用该类型对应的 Class 实例对象的方法。这样一个过程，就是把一个类型的二进制数据解析为方法区的内部数据结构，并在堆上建立一个 Class 对象的过程，这被称为“创建”类型。

Java 类型要么由启动类装载器装载，要么通过用户自定义的类装载器装载。启动类装载器是虚拟机实现的一部分，它以与实现无关的方式装载类型(包括 JavaAPI 的类和接口)，用户自定义的类装载器是类 java.lang.ClassLoader 的子类实例，它以定制的方式装入类。

类装载器(启动型或者用户自定义的)并不需要一直等到某个类型“首次主动使用”时再去装入它。Java 虚拟机规范允许类装载器缓存 Java 类型的二进制表现形式，在预料某个类型将要被使用时就装载它，或者把这些类型装载到一些相关的分组里面。如果一个类装载器在预先装载时遇到问题，无论如何，它应该在该类型被首次主动使用时报告该问题(通过抛出一个 LinkageError 异常的子类)。换句话说，如果一个类装载器在预先装载时遇到缺失或者错误的 class 文件，它必须等到程序首次主动使用该类时才报告错误。如果这个类一直没有被程序主动使用，那么该类装载器将不会报告错误。

7.1.2 验证

当类型被装载后，就准备连接了。连接过程的第一步是验证——确认类型符合 Java 语言的语义，并且它不会危及虚拟机的完整性。

在验证上，不同的虚拟机实现拥有一些灵活性。虚拟机实现的设计者可以决定如何以及何时验证类型。Java 虚拟机规范列出了虚拟机可以抛出的异常以及在何种条件下必须抛出它们。不管 Java 虚拟机可能遇到了什么样的麻烦，都应该有一个异常或者错误可以抛出。规范表明了在每种情形下应该抛出何种异常或者错误。某些情况下，规范明确地说明何时这种异常或者错误应该被抛出，但是通常没有严格地规定应该如何或者在何时检查错误条件。

不管怎样，在大多数 Java 虚拟机实现中特定类型的检查一般都在特定的时间发生。比如，在装载过程中，虚拟机必须解析代表类型的二进制数据流，并且构造内部数据结构。这个时候，必须做一些特定的检查，以保证解析二进制数据的初始工作不会导致虚拟机奔溃。在这个解析期间，虚拟机大多会检查二进制数据以确保数据全部是预期的格式。Javaclass 文件格式的解析器可能检查魔数，确保每一个部分都在正确的位置，拥有正确的长度，验证文件不是太长或者太短，等等。虽然这些检查在装载期间完成，实在正式的连接验证阶段之前进行，但它们仍然在逻辑上属于验证阶段。检查被装载的类型是否有任何问题的整个过程都属于验证。

另外一个很可能在装载时进行的检查是，确保除了 Object 之外的每一个类都有一个超类。在装载时检查的原因是当虚拟机装载一个类时，它必须确保该类的所有超类都已经被装载了。对于给定的类，得到其超类名字的唯一方法就是观察类的二进制数据。因为 Java 虚拟机无论如何都要在装载的时候检查每个类的超类数据，所以在装载阶段做这个检查是顺理成章的。

在大部分虚拟机实现中，还有一种检查往往发生在正式的验证阶段之后，那就是符号引用的验证。动态连接的过程包括通过保存在常量池中的符号引用，这些被引用的包括类、接口、字段以及方法，这个阶段需要把符号引用替换成直接引用。当虚拟机搜寻一个被符号引用的元素时，它必须首先确认该元素存在。如果虚拟机发现元素存在，它必须进一步检查引用类型有访问元素的权限。这些对存在性和访问权限的检查逻辑上是验证的一部分，属于连接的第一阶段，但是往往在解析的时候发生，那是连接的第三阶段。解析自身也可能延迟到符号引用第一次被程序所使用时，所以这些检查甚至有可能在初始化之后才进行。

那么在正式的验证阶段做哪些检查呢？任何在此之前还没有进行的检查以及在此之后不会被检查的项目都包含在内。在此首先需要列出确保各个类之间二进制兼容的检查。

- 检查 final 的类不能拥有子类。
- 检查 final 的方法不能被覆盖。
- 确保在类型和超类型之间没有不兼容的方法声明(比如两个方法拥有同样的名字，参数在数量顺序、类型上都相同，但是返回类型不同)。

请注意，当这些检查需要查看其他类型的时候，它只需要查看超类型。超类需要在子类初始化前被初始化，所以这些类应该已经被装载了。当实现了父接口的类被初始化的时候，不需要初始化父接口。然而，当实现了父接口的子类(或者是扩展了父接口的子接口)被装载时，父接口也必须被装载。它们不会被初始化，只是被装载了，可能被某些虚拟机实现可选地连接了。当装载一个类的时候，它所有的超类都会被装载。在验证期间，这个类和它所有的超类型都需要确保互相之间仍然二进制兼容。

- 检查所有的常量池入口相互之间一致。比如，一个 CONSTANT_String_info 入口的 string_index 项目必须是一个 CONSTANT_Utf8_info 入口的索引。
- 检查常量池中的所有的特殊字符串，例如检查类名、字段名和方法名、字段描述符和方法描述符是否符合格式。
- 检查字节码的完整性。

上面列出的最复杂的任务就是字节码验证。所有的 Java 虚拟机都必须设法为它们执行的每个方法检验字节码的完整性。比如，不能因为一个超出了方法末尾的跳转指令就导致虚拟机实现崩溃。它们必须在字节码验证的时候检查出这样的跳转指令是非法的，从而抛出一个错误。

虚拟机的实现没有强求在正式的连接验证阶段进行字节码验证。所有的 Java 虚拟机都必须设法为它们执行的每个方法验证字节码的完整性。比如，不能因为一个超出了方法末尾的跳转指令就导致虚拟机实现崩溃。它们必须在字节码验证的时候检查出这样的跳转指令是非法的，从而抛出一个错误。

虚拟机的实现没有强求在正式的连接验证阶段进行字节码验证。比如，实现可以自由地选择在执行每条语句的时候单独进行验证。然而，Java 虚拟机指令集设计的一个目标就是使得字节码流可以通过一次性使用一个数据流分析器进行验证。在连接过程中一次性验证字节码流，而非在程序执行的时候动态验证，使得 Java 程序的运行速度得到很大的提高。

当通过一个数据流分析器进行字节码验证的时候，虚拟机可能不得不为了确保符合 Java 语言的语义而装载其他的类。比如，设想一个类包含了一个方法，其中把一个 Java.lang 的实例的引用赋值给了一个 java.lang.Number 类型的字段。在这个情况下，虚拟机将在字节码验证的时候装载类 Float。确保这是一个 Number 类的子类。它也不得不装载 Number 来确保它没有被声明为 final。虚拟机此时不需要初始化 Float，只需要装载它，Float 会在首次主动使用时被初始化。

7.1.3 准备

随着 Java 虚拟机装载了一个类，并执行了一些它选择进行的验证之后，类就可以进入准备阶段了。在准备阶段，Java 虚拟机为类变量分配内存，设置默认初始值。但在到达初始化阶段之前，类变量都没有被初始化为真正的初始值。在准备阶段是不会执行 Java 代码的，此阶段虚拟机会把给类标量新分配的内存根据类型设置默认值。在此需要注意 boolean 类型，Java 虚拟机不太支持这一类型。在内部的 boolean 常常被实现为一个 int，会被默认地设置为 0，也就是 boolean 取 false 值。因此 boolean 类变量，就算他们在内部是被作为 int 实现的，也总是被初始化成 false。

在准备阶段，Java 虚拟机实现可能也为一些数据结构分配内存，目的是提高运行程序的性能。这种数据结构的实例如方法表，它包含指向类中每一个方法(包括从超类继承的方法)的指针。方法表可以使得继承的方法执行时不需要搜索超类。

7.1.4 解析

当类型经过验证和准备这两个阶段之后，就可以进入第三个(也就是最后一个)连接阶段：解析。解析过程就是在类型的常量池中寻找类、接口、字段和方法的符号引用，把这些符号引用替换成直接引用的过程。

7.1.5 初始化

在 Java 代码中，一个正确的初始值是通过类变量初始化语句或者静态初始化语句给出的。

1. <clinit>方法

所有的类变量初始化语句和类型的静态初始化器都被 Java 编译器收集在一起，放到一个特殊的方法中，称为类初始化方法。在类和接口的 Java Class 文件中，这个方法被称为”<clinit>”。通常的 Java 程序方法是无法调用这个<clinit>方法的，只能被 Java 虚拟机调用。

初始化一个类的过程包含以下两个步骤：

(1) 如果类存在直接超类的话，且直接超类还没有被初始化，则先初始化直接超类。

(2) 如果类存在一个类初始化方法(<clinit>)就执行此方法。

只需一步即可初始化一个接口：如果接口存在一个接口初始化方法的话，就执行此方法。初始化的顺序按照类变量初始化语句和静态初始化语句出现的顺序初始化。

并不是所有的类都需要在它的 class 文件中有一个<clinit>()方法。如果类没有声明任何类变量，也没有静态初始化语句，那么它就不会有<clinit>()方法。如果类声明了类变量，但是没有明确使用类变量初始化语句或者静态初始化语句初始化它们，那么类也不会有<clinit>()方法。

所有在接口中声明的隐式公开(Publict)、静态(Static)、最终(Final)字段必须在字段初始化语句中初始化，如果接口包含任何不能在编译时被解析成为一个常量的字段初始化语

句，接口就拥有一个<clinit>()方法。例如：

```
interface Example{
     int  ketchup = 5;
   int mustard= (int)  (Math.random()*5.0);
}
```

ketchup 将会被初始化为一个编译时常量，而 mustard 字段被方法<clinit>()初始化。

2．主动使用与被动使用

当使用一个非常量的静态字段时，只有这个字段是被当前类或接口声明的情况下才是主动使用，如果是子类使用父类中声明的字段，子接口和实现了该接口的类使用此接口中的字段都被认为是被动使用，不会引发初始化，例如：

```
public class NewParent {
   static int hoursOfsleep = (int)(Math.random()*3.0);
   static{
      System.out.println("NewParentwas initialized");
   }
}
public class NewbornBaby extends NewParent {
   static int housOfCrying = 6+(int)(Math.random()*2.0);
   static{
      System.out.println("NewbornBabywas initialized.");
   }
}
public class Example {
   static{
      System.out.println("Example wasinitialized.");
   }
   public static void main(String[] args) {
      int hours = NewbornBaby.hoursOfsleep;
      System.out.println(hours);
   }
}
```

运行结果：

```
Example was initialized.
NewParent was initialized
2
```

7.2　对象的生命周期

一旦一个对象被装载、连接并初始化之后，就可以随时使用它了。程序可以访问它的静态字段，调用它的静态方法，或者创建它的实例。在 Java 程序中，类可以被明确或者隐含的实例化。实例化一个类有 4 种途径：明确的使用 new 操作符，调用 Class 或者 java.lang.reflect.Construtor 对象的 newInstance()方法，调用任何现有对象的 clone()方法，或者通过 java.io.ObjectInputStream 类的 getObjet()方法反序列化。

在下面的代码中演示了其中三种创建新的类实例的方法：

```
public class Example4 implements Cloneable {
     Example4(){
          System.out.println("Createdby invoking newInstance()!");
     }
     Example4(Stringmsg) {
          System.out.println(msg);
     }
     public static void main(String[]args) throws ClassNotFoundException,
InstantiationException,
               IllegalAccessException,CloneNotSupportedException {
          Example4e4 = new Example4("created with new!");
          ClassmycClass = Class.forName("Example4");
          Example4obj2 = (Example4) mycClass.newInstance();
          Example4obj3 = (Example4) obj2.clone();
     }
}
```

运行后会打印出如下输出：

```
created with new!
Created by invokingnewInstance()!
```

除了这 4 种在 Java 源代码中明确地实例化对象的方法之外，还有几种情况下对象会被隐含地实例化。

在任何 Java 程序中第一个隐含实例化对象可能就是保存命令行参数的 String 对象。每一个命令行参数都会有一个 String 对象的引用，把它们组成一个 String 数组并作为一个参数传递到每一个程序的 main 方法中。

另外两种隐含实例化类的方法和类装载的过程有关。首先，对于 Java 虚拟机装载的每一个类型来说，它会暗中实例化一个 Class 对象来代表这个类型。其次，当 Java 虚拟机装载了在常量池中包含 CONSTANT_String_info 入口的类时，它会创建新的 String 对象的实例来表示这些常量字符串。把方法区中的 CONSTANT_String_info 入口转换成一个堆中的 String 实例的过程是常量池解析过程的一部分。

还有一条隐含创建对象的途径是通过执行包含字符串操作符的表达式产生对象。如果这样的字符串不是一个编译时常量，用于中间处理的 String 和 StringBuffer 对象会在计算表达式的过程中创建。下面是一个例子。

```
class Example5 {
     public static void main(String[]args) {
          if (args.length < 2) {
               System.out.println("mustenter any two arg");
               return;
          }
          System.out.println(args[0]+ args[1]);
     }
}
```

Javac 为上述 Example5 的 main()方法产生了下面的字节码：

```
0 aload_0                    //pushthe objref from loc var 0 (args)
```

```
1 arraylength            //poparrayref, calc array length,push int length
2 iconst 2              //push2 ints, compare, branch if (length>=2) to
3 if icmpge 15          //offset 15
                        //Push objref of string literal
6 getstatic #11 <Field java.io.PrintStreamout>
                        //Push objref of string literal
9 ldc #1 <String "must enter any two args">
                         //Pop objref to String param,objref to
System.out, invoke println()
11 invokevirtual #12 <Method voidprintln(java.lang.String)>
14 return               //Return void from main()
                        //Push objref from System.out
15 getstatic #11 <Field java.io.PrintStreamout>
                        //The string concatenation operation begins here
                        //Allocate mem for new StringBuffer object, and
initialize mem to default initialvalues, push objref to new object.
18 new #6 <Class java.lang.StringBuffer>
21 dup                 //Duplicate objref to StringBuffer object
22 aload 0             //Push ref from loc var 0 (args)
23 iconst 0            //Push int constant 0
                       //Pop int, arrayref, push String at arrayref[int]
24 aaload              //which is args[0]
                       //Pop objref, invoke Stirng's class method
valueof(), passing it the objref tothe args[0] String object. Valueof()
calls toString() on the ref, and returns(and pushes) the result, which
happens to be the original args[0] String. Inthis case, the stack will
look precisely the same.
```

在上述 Example5 的 main()方法的字节码中，包含了三个隐含创建的 String 对象和一个 StringBuffer 对象。其中两个 String 对象的引用作为传递到 main()方法的 args 数组的一部分，是通过位于偏移量为 24 和 33 的 aload 指令压入栈的。StringBuffer 是在偏移量为 18 的 new 指令创建的，被偏移量为 28 的 invokespecial 指令初始化。最后一个 String 对象代表 args[0]和 args[1]的连接，是通过调用 StringBuffer 对象的 toString()方法建立的，这是由位于偏移量为 37 的 invokevirtual 指令完成的。

当 Java 虚拟机创建一个类的新实例时，不管是明确的还是隐含的，首先都需要在堆中为保存对象的实例变量分配内存。所有在对象的类中和它的超类中声明的变量(包括隐藏的实例变量)都要分配内存。堆中对象的映像中其他一些与实现相关的元素，比如指向方法区中类数据的指针，大致也是在这个时间分配的。一旦虚拟机为新的对象准备好了堆内存，它立即把实例变量初始化为默认的初始值。

一旦虚拟机完成了为新对象分配内存和为实例变量赋默认初始值后，它随后就会为实例变量赋正确的初始值。根据创建对象的方法不同，Java 虚拟机使用三种技术之一来完成这个工作。如果对象是通过 clone()调用来创建的，虚拟机把原来被克隆的实例变量中的值复制到新对象中。如果对象是通过一个 ObjectInputStream 的 readObject()调用反序列化的，虚拟机通过从输入流中读入的值来初始化那些非暂时性的实例变量。否则，虚拟机调用对象的实例初始化方法。实例初始化方法把对象的实例变量初始化为正确的初始值。

Java 编译器为它编译的每一个类都至少生成一个实例初始化方法。在 Java 的 class 文件中，这个实例初始化方法被称为“<init>”。针对源代码中每一个类的构造方法，Java

编译器都产生一个<init>()方法。如果类没有明确地声明任何构造方法，编译器默认产生一个无参数的构造方法，它仅仅调用超类的无参数构造方法。和其他的构造方法一样，编译器在 class 文件中创建一个<init>()方法，对应它的默认构造方法。

一个<init>()方法中可能包含三种代码：调用另一个<init>()方法，实现对任何实例变量的初始化，构造方法体的代码。如果构造方法通过明确地调用同一个类中的另一个构造方法(一个 this()调用)开始，它对应的<init>()方法由两部分组成：

- 一个同类的<init>()方法的调用，任意实例变量初始化方法的字节码；
- 一个超类的<init>()方法的调用，实现了对应构造方法的方法体的字节码。

如果构造方法没有使用 this()调用开始，并且这个类是 Object，上面类表中的第一个元素就不存在。因为 Object 没有超类，它的<init>()方法就不能通过调用超类的<init>()方法开始。

如果构造方法通过明确地调用超类的构造方法(一个 super()调用)开始，它的<init>()方法会调用对应的超类的<init>()方法。比如，如果一个构造方法通过明确地调用“super(int,String)构造方法”开始，对应的<init>()方法会从调用超类的“<init>(int, String)”方法开始。如果构造方法没有明确地从 this()或者 super()调用开始，对应的<init>()方法默认会调用超类的无参数<init>()方法。

下面的例子包含了三个构造方法，编号从 1 到 3：

```
public class Example6 {
     private int width = 3;
     // constructorone
     // Thisconstructor begins with a this() constructor invocation
     // which getscompiled to a same-class <init>() method incocation
     Example6(){
          this(1);
          System.out.println("Example6(),width=" + width);
     }
     // constructortwo
     // Thisconstructor begins with no explicit invocation of another
     // constructor
     // so it willget compiled to an <init>() method that begins with an
     // invocationof the
     // superclass's no-arg<init>() method.
     Example6(int width) {
          this.width = width;
          System.out.println("Example6(int),width=" + width);
     }
     //
     Example6(Stringmsg) {
          super();
          System.out.println("Example6(String),width=" + width);
          System.out.println(msg);
     }
     public static void main(String[]args) {
          Stringmsg = "The agapanthus is also know as Lily of the Nile";
          Example6one = new Example6();
          Example6two = new Example6(2);
          Example6three = new Example6(msg);
```

```
    }
}
```

执行后会打印如下输出：

```
Example6(int), width=1
Example6(), width=1
Example6(int), width=2
Example6(String), width=3
The agapanthus is also know as Lily of the Nile
```

7.3　卸载类型

在很多方面，Java 虚拟机中类的生命周期和对象的生命周期很相似。虚拟机创建并初始化对象，使程序能使用对象，然后在对象变得不再被引用后可选地执行垃圾收集。同样，虚拟机装载，连接并初始化类，使程序能使用类，当程序不再引用它们的时候可选地卸载它们。

7.3.1　卸载类型基础

类的垃圾收集和卸载之所以在 Java 虚拟机中很重要，是因为 Java 程序可以在运行时通过用户自定义的类装载器装载类型来动态地扩展程序。所有被装载的类型都在方法区占据内存空间。如果 Java 程序持续通过用户自定义的类装载器装载类型，方法区的内存足迹就会不断增长。如果某些动态装载的类型只是临时需要，当它们不再被引用之后，占据的内存空间可以通过卸载类型而释放。

Java 虚拟机通过何种方法来确定一个动态装载的类型是否仍然被程序需要呢，其判断方式与判断对象是否仍然被程序需要的方式很类似。如果程序不再引用某类型，那么这个类型就无法再对未来的计算过程产生影响。类型变成不可触及的，而且可以被垃圾收集。

使用启动类装载器装载的类型永远是可触及的，所以永远不会被卸载。之后使用用户定义的类装载器装载的类型才会变成不可触及，从而被虚拟机回收。如果某个类型的 Class 实例被发现无法通过正常的垃圾收集堆触及，那么这个类型就是不可触及的。

判断动态装载类型的 Class 实例在正常的垃圾收集过程中是否可以触及有如下两种方式。

- 第一种：是最明显的一种方式，如果程序保持对 Class 实例的明确引用，它就是可触及的。
- 第二种：如果在堆中还存在一个可触及的对象，在方法区中它的类型数据指向一个 Class 实例，那么这个 Class 实例就是可触及的。

7.3.2　unreachable 状态的作用

在 Java 虚拟机规范中，关于类型卸载的描述含义是：只有当加载该类型的类加载器实例(非类加载器类型)为 unreachable 状态时，当前被加载的类型才被卸载。启动类加载器实

例永远为 reachable 状态，由启动类加载器加载的类型可能永远不会被卸载。

由此可以看出，类型卸载(Unloading)仅仅是作为一种减少内存使用的性能优化措施存在的，具体和虚拟机实现有关，对开发者来说是透明的。

纵观 Java 语言规范及其相关的 API 规范，找不到显示类型卸载的接口，也就是说：

(1) 一个已经加载的类型被卸载的几率很小，至少被卸载的时间是不确定的；

(2) 一个被特定类加载器实例加载的类型运行时可以认为是无法被更新的。

因为如果想卸载某类型，必须保证加载该类型的类加载器处于 unreachable 状态。所以从某种程度上讲，在一个稍微复杂的 Java 应用中，我们很难准确判断出一个实例是否处于 unreachable 状态。接下来为了更加准确的逼近这个所谓的 unreachable 状态，我们用一个简单的演示代码来测试。

请看第一个测试：使用自定义类加载器加载，然后测试将其设置为 unreachable 的状态。我们自定义类加载器，为了简单起见，这里就假设加载当前工程以外 D 盘某文件夹的 class。然后假设目前有一个简单自定义类型 MyClass 对应的字节码存在于 D:/classes 目录下。

```
public class MyURLClassLoader extends URLClassLoader {
   public MyURLClassLoader() {
      super(getMyURLs());
    }
   private static URL[] getMyURLs() {
    try {
       return new URL[]{new File ("D: /classes/").toURL()};
     } catch (Exception e) {
        e.printStackTrace();
       return null;
     }
    }
}
```

下面是简单的测试代码：

```
public class Main {
     public static void main(String[] args) {
       try {
           MyURLClassLoader classLoader = new MyURLClassLoader();
           Class classLoaded = classLoader.loadClass("MyClass");
           System.out.println(classLoaded.getName());
           classLoaded = null;
           classLoader = null;
           System.out.println("开始 GC");
           System.gc();
           System.out.println("GC 完成");
         } catch (Exception e) {
             e.printStackTrace();
         }
      }
 }
```

然后增加虚拟机参数-verbose:gc 来观察垃圾收集的情况，下面是对应的输出：

```
MyClass
```

```
开始 GC...
[Full GC[Unloading class MyClass]
207K->131K(1984K), 0.0126452 secs]
GC 完成.
```

接下来看第二个测试：使用系统类加载器加载，但是无法将其设置为 unreachable 的状态。在此将场景一中的 MyClass 类型字节码文件放置到工程的输出目录下，以便系统类加载器可以加载。

```
public class Main {
    public static void main(String[] args) {
     try {
       Class classLoaded =
ClassLoader.getSystemClassLoader().loadClass(
 "MyClass");

System.out.printl(sun.misc.Launcher.getLauncher().getClassLoader());
      System.out.println(classLoaded.getClassLoader());
      System.out.println(Main.class.getClassLoader());
      classLoaded = null;
      System.out.println("开始 GC");
      System.gc();
      System.out.println("GC 完成");
     //判断当前系统类加载器是否有被引用(是否是 unreachable 状态)
      System.out.println(Main.class.getClassLoader());
     } catch (Exception e) {
        e.printStackTrace();
     }
   }
 }
```

和前面的第一个测试一样，通过增加虚拟机参数-verbose:gc 来观察垃圾收集的情况，下面是对应的输出：

```
sun.misc.Launcher$AppClassLoader@197d257
sun.misc.Launcher$AppClassLoader@197d257
sun.misc.Launcher$AppClassLoader@197d257
开始 GC...
[FullGC196K->131K(1984K), 0.0130748 secs]
GC 完成...
sun.misc.Launcher$AppClassLoader@197d257
```

由于系统 ClassLoader 实例(AppClassLoader@197d257">sun.misc.Launcher$AppClassLoader@197d257)加载了很多类型，而且又没有明确的接口将其设置为 null，所以不能将加载 MyClass 类型的系统类加载器实例设置为 unreachable 状态。通过测试结果可以看出，MyClass 类型并没有被卸载，这说明像类加载器实例这种较为特殊的对象一样，一般在很多地方被引用，并且会在虚拟机中呆比较长的时间。

接下来看第三个测试：使用扩展类加载器加载，但是无法将其设置为 unreachable 的状态。在此将前面第二个测试中的 MyClass 类型字节码文件打包成 jar 放置到 JRE 扩展目录下，这样可以方便扩展类加载器可以加载的到。因为下面的标志扩展 ClassLoader 实例加载了很多类型：

```
ExtClassLoader@7259da">sun.misc.Launcher$ExtClassLoader@7259da
```

并且又没有明确的接口将其设置为 null，所以不能将加载 MyClass 类型的系统类加载器实例设置为 unreachable 状态，所以通过测试结果我们可以看出，MyClass 类型并没有被卸载。

```
public class Main {
     public static void main(String[] args) {
       try {
          Class classLoaded =
ClassLoader.getSystemClassLoader().getParent()
 .loadClass("MyClass");
          System.out.println(classLoaded.getClassLoader());
          classLoaded = null;
          System.out.println("开始 GC");
          System.gc();
          System.out.println("GC 完成");
         //判断当前标准扩展类加载器是否有被引用(是否是 unreachable 状态)
          System.out.println(Main.class.getClassLoader().getParent());
       } catch (Exception e) {
          e.printStackTrace();
       }
    }
 }
```

接下来同样增加虚拟机参数-verbose:gc 来观察垃圾收集的情况，会看到下面的输出：

```
sun.misc.Launcher$ExtClassLoader@7259da
开始 GC...
[FullGC199K->133K(1984K), 0.0139811 secs]
GC 完成...
sun.misc.Launcher$ExtClassLoader@7259da
```

通过上述介绍的三个相关测试，针对卸载类型可以作出如下三个总结。

(1) 在整个运行期间，有启动类加载器加载的类型是不可能被卸载的。

(2) 在运行期间，被系统类加载器和标准扩展类加载器加载的类型不太可能被卸载，因为系统类加载器实例或者标准扩展类的实例基本上在整个运行期间总能直接或者间接的访问的到，其达到 unreachable 的可能性极小。但是在虚拟机快退出的时候可以，因为无论 ClassLoader 实例或者 Class(java.lang.Class)实例也都是在堆中存在，同样遵循垃圾收集的规则。

(3) 被开发者自定义的类加载器实例加载的类型只有在很简单的上下文环境中才能被卸载，而且一般还要借助于强制调用虚拟机的垃圾收集功能才可以做到。在稍微复杂点的应用场景中，尤其很多时候用户在开发自定义类加载器实例的时候，可以采用缓存的策略以提高系统性能。被加载的类型在运行期间也是几乎不太可能被卸载的，至少卸载的时间是不确定的。

由此可见，一个已经加载的类型被卸载的几率很小至少被卸载的时间是不确定的。同时可以看出，开发者在开发代码的时候，不应该对虚拟机的类型卸载做任何假设的前提下来实现系统中的特定功能。

7.3.3 类型更新

在运行时被一个特定类加载器实例加载的特定类型是无法被更新的，这里说的是一个特定的类加载器实例，而并不是一个特定的类加载器类型。接下来请读者看第四个测试场景：删除前面已经放在工程输出目录下和扩展目录下的对应的 MyClass 类型对应的字节码。

```
public class Main {
      public static void main(String[] args) {
        try {
          MyURLClassLoader classLoader = new MyURLClassLoader();
          Class classLoaded1 = classLoader.loadClass("MyClass");
          Class classLoaded2 = classLoader.loadClass("MyClass");
          //判断两次加载 classloader 实例是否相同
           System.out.println(classLoaded1.getClassLoader() ==
classLoaded2.getClassLoader());
        //判断两个 Class 实例是否相同
          System.out.println(classLoaded1 == classLoaded2);
      } catch (Exception e) {
          e.printStackTrace();
      }
   }
 }
```

执行后会输出如下结果：

```
true
true
```

通过上述结果可以看出，两次加载获取到的两个 Class 类型实例是相同的，基于此，是不是可以确定是我们的自定义类加载器真正意义上加载了两次呢？通过对 java.lang.ClassLoader 的 loadClass(String name，boolean resolve)方法进行调试，可以看出第二次加载并不是真正意义上的加载，而是直接返回了上次加载的结果。

第五个测试场景：同一个类加载器实例重复加载同一类型。首先对已有的用户自定义类加载器做一定的修改，覆盖已有的类加载逻辑，修改类 MyURLClassLoader.java 之后的代码如下：

```
public class MyURLClassLoader extends URLClassLoader {
     //省略部分的代码和前面相同，只是新增如下覆盖方法
     /*
     * 覆盖默认的加载逻辑，如果是 D：/classes/下的类型每次强制重新完整加载
     * @see java.lang.ClassLoader#loadClass(java.lang.String)
     */
     @Override
     public Class<?> loadClass(String name) throws
ClassNotFoundException {
     try {
       //首先调用系统类加载器加载
        Class c = ClassLoader.getSystemClassLoader().loadClass(name);
       return c;
```

```
    } catch (ClassNotFoundException e) {
      // 如果系统类加载器及其父类加载器加载不上，则调用自身逻辑来加载 D：/classes/下的类型
        return this.findClass(name);
      }
    }
  }
```

在上述代码中，this.findClass(name)会进一步调用父类 URLClassLoader 中的对应方法，其中涉及到了 defineClass(String name)的调用，所以说现在类加载器 MyURLClassLoader 会针对“D:/classes/”目录下的类型进行真正意义上的强制加载并定义对应的类型信息。此时重新运行第四个测试场景代码后会输出：

```
Exception in thread "main" java.lang.LinkageError： duplicate class
definition： MyClass
at java.lang.ClassLoader.defineClass1(Native Method)
at java.lang.ClassLoader.defineClass(ClassLoader.java：620)
at java.security.SecureClassLoader.defineClass(SecureClassLoader.java：
124)
at java.net.URLClassLoader.defineClass(URLClassLoader.java：260)
at java.net.URLClassLoader.access$100(URLClassLoader.java：56)
at java.net.URLClassLoader$1.run(URLClassLoader.java：195)
at java.security.AccessController.doPrivileged(Native Method)
at java.net.URLClassLoader.findClass(URLClassLoader.java：188)
at MyURLClassLoader.loadClass(MyURLClassLoader.java：51)
at Main.main(Main.java：27)
```

由此可以看出，如果同一个类加载器实例重复强制加载(含有定义类型 defineClass 动作)相同类型，会引起 java.lang.LinkageError: duplicate class definition。

第六个测试场景：同一个加载器类型的不同实例重复加载同一类型。

```
public class Main {
    public static void main(String[] args) {
      try {
        MyURLClassLoader classLoader1 = new MyURLClassLoader();
        Class classLoaded1 = classLoader1.loadClass("MyClass");
        MyURLClassLoader classLoader2 = new MyURLClassLoader();
        Class classLoaded2 = classLoader2.loadClass("MyClass");
        //判断两个 Class 实例是否相同
        System.out.println(classLoaded1 == classLoaded2);
      } catch (Exception e) {
        e.printStackTrace();
      }
    }
  }
```

此时执行后会输出下面的内容：

```
false
```

由此可见，由不同类加载器实例重复强制加载(含有定义类型 defineClass 动作)同一类型不会引起 java.lang.LinkageError 错误，但是加载结果对应的 Class 类型实例是不同的，即实际上是不同的类型(虽然包名+类名相同)。如果强制转化使用，会引起

ClassCastException。

我们在开发的时候可能会遇到这样的需求，就是要动态加载某指定类型 Class 文件的不同版本，以便能动态更新对应功能。此时建议大家不用等待指定类型的以前版本被卸载，因为卸载行为对 Java 开发人员是透明的。比较可靠的做法是，每次创建特定类加载器的新实例来加载指定类型的不同版本，这种使用场景下，一般就要牺牲缓存特定类型的类加载器实例以带来性能优化的策略了。对于指定类型已经被加载的版本，会在适当时机达到 unreachable 状态，被 unload 并垃圾回收。每次使用完类加载器特定实例后(确定不需要再使用时)，将其显示为 null，这样可能会比较快的达到 JVM 规范中所说的类加载器实例 unreachable 状态，增大已经不再使用的类型版本被尽快卸载的机会。

当每次用新的类加载器实例去加载指定类型的指定版本，确实会带来一定的内存消耗，一般类加载器实例会在内存中保留比较长的时间。

7.4　常量入栈操作

因为 Java 虚拟机是基于栈的机器，所以几乎所有 Java 虚拟机的指令都与操作数栈相关。栈操作包括把常量压入操作数栈、执行通用的栈操作、在操作数栈和局部变量之间往返传输值。

和栈操作相关的基本指令如下：

- store：表示弹出操作数栈(操作数栈是一个栈)顶的数据放入局部变量区；
- store_x：表示弹出操作数栈顶的数据放入局部变量区索引为 x 的地方；
- load：表示将局部变量区中某个位置(即某个索引，因为局部变量区是一个数组)的局部变量压入操作数栈；
- load_x：表示将局部变量区中 x 位置的局部变量压入操作数栈；
- astore：表示弹出操作数栈顶的对象引用，并放入局部变量区；
- astore_x：也跟前面有相同的规则；
- aload：表示将局部变量区中某个位置的对象引用压入操作数栈；
- aload_x：也跟前面有相同的规则；
- const_x：表示将某个值(x)压入操作数栈；
- bipush x：表示将某个值(x，类型是 byte)转换为 int 类型压入操作数栈；
- sipush x 表示将某个值(x，类型为 short)转换为 int 类型压入操作数栈；
- ldc x：表示将常量池中的某个入口地址(x 表示常量池入口地址)压入操作数栈；
- pop：表示将操作数栈顶部的数据弹出栈；
- dup：表示复制操作数栈顶的数据。

为了演示上述指令的作用，我们可以使用 jClassLib 工具来配合，这是一个开源的分析类文件内容的工具，可以从网上下载，运行之后的界面如图 7-1 所示。

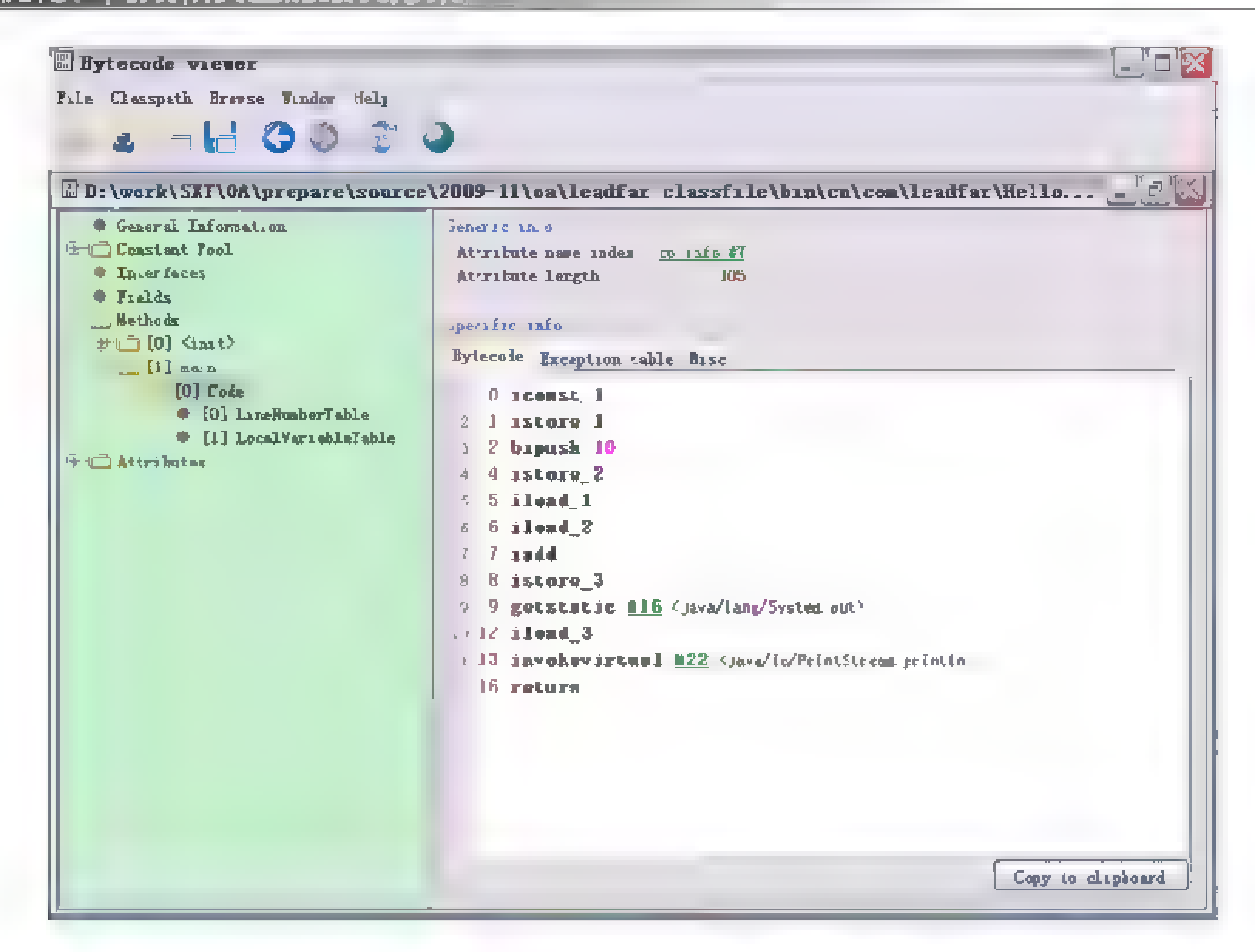


图 7-1 jClassLib 工具的界面

操作码在执行常量入栈操作之前，使用三种方式指明常量的值：常量值隐含在操作码内部，常量值在字节码中如同操作数一样跟随在操作码之后，或者从常量池中取出常量。

将一个字长的常量压入栈的操作说明如表 7-1 所示。

表 7-1 将一个字长的常量压入栈的操作说明

操 作 码	操 作 数	说 明
iconst_ml	(无)	将 int 类型值-1 压入栈
iconst_0	(无)	将 int 类型值 0 压入栈
iconst_1	(无)	将 int 类型值 1 压入栈
iconst_2	(无)	将 int 类型值 2 压入栈
iconst_3	(无)	将 int 类型值 3 压入栈
iconst_4	(无)	将 int 类型值 4 压入栈
iconst_5	(无)	将 int 类型值 5 压入栈
fconst_0	(无)	将 float 类型值 0 压入栈
fconst_1	(无)	将 float 类型值 1 压入栈
fconst_2	(无)	将 float 类型值 2 压入栈

将两个字长的常量压入栈的操作说明如表 7-2 所示。

表 7-2 将两个字长的常量压入栈的操作说明

操 作 码	操 作 数	说 明
lconst_0	(无)	将 long 类型值 0 压入栈
lconst_1	(无)	将 long 类型值 1 压入栈
dconst_0	(无)	将 double 类型值 0 压入栈
dconst_1	(无)	将 double 类型值 1 压入栈

如果给一个对象引用赋空值时会用到 aconst_null 指令，将空(null)对象引用压入栈的说明如表 7-3 所示。

表 7-3　将空(null)对象引用压入栈的操作说明

操 作 码	操 作 数	说　明
aconst_null	(无)	将空(null)对象引用压入栈

请看下面的演示代码：

```
public class HelloWorld01 {
    /**
     * @param args
     */
    public static void main(String[] args) {
       int i0 = 1;
       int i1 = 10;
       int i2 = i0 + i1;
       System.out.println(i2);
    }
}
```

对应的方法的指令如下：

```
 0 iconst_1
 1 istore_1
 2 bipush 10
 4 istore_2
 5 iload_1
 6 iload_2
 7 iadd
 8 istore_3
 9 getstatic #16 <java/lang/System.out>
12 iload_3
13 invokevirtual #22 <java/io/PrintStream.println>
16 return
```

上述指令的具体说明如下：

- int i0=1：目标是给变量 i0 赋值；
- iconst_1：意思是将 1 这个值压入操作数栈；
- istore_1：意思是将操作数栈顶的数据(即刚压进去的值 1)弹出并存储在变量区中索引为 1 的地方；
- bipush 10：意思是将 10 压入操作数栈；
- istore_2：意思是将操作数栈顶的数据(即刚压进去的 10)弹出并存储在变量区中索引为 2 的地方；
- iload_1：表示将变量区中索引为 1 的值压入操作数栈(这个值就是 1)；
- iload_2：表示将变量区中索引为 2 的值压入操作数栈(这个值就是 10)；
- iadd：表示将操作数栈中的两个数弹出相加，并将结果压入操作数栈中；
- istore_3：表示将操作数栈顶的数据(即在 iadd 操作码中压进去的相加之后的结果)弹出并存储在变量区中索引为 3 的地方。

从上面的分析可以看到，例如 1 和 10 之类常量值，编译器可以直接将它们放在指令序列中。

请读者再看下面的代码：

```
public class HelloWorld02 {
    /**
     * @param args
     */
    public static void main(String[] args) {
        long i0 = 100;
        long i1 = 999999999L;
        long i2 = i0 + i1;
        System.out.println(i2);
    }
}
```

对应的指令序列为：

```
 0 ldc2_w #16 <100>
 3 lstore_1
 4 ldc2_w #18 <999999999>
 7 lstore_3
 8 lload_1
 9 lload_3
10 ladd
11 lstore 5
13 getstatic #20 <java/lang/System.out>
16 lload 5
18 invokevirtual #26 <java/io/PrintStream.println>
21 return
```

上述指令的具体说明如下：

- ldc2_w #16：表示将常量池 16 号的值压入操作数栈，两个字长(一个字长是 32 位)的宽度；
- lstore_1：表示将操作数栈顶的数据弹出放到变量区索引号为 1 的地方；
- ldc2_w #18：表示将常量池 18 号的值压入操作数栈；
- lstore_3：表示将操作数栈顶的数据弹出放到变量区索引号为 3 的地方。之所以放到索引号为 3 而不是 2 的地方，是因为一个 Long 占据了两个字长，即两个索引号；
- lload_1 和 lload_3：表示将变量区索引号为 1 和 3 的两个值压入操作数栈；
- ladd：表示将操作数栈中的两个数据弹出并相加，结果再次入栈；
- lstore 5：表示将操作数栈顶的数据(即刚才相加的计算结果)弹出，并放到变量区索引号为 5 的地方。

从上面的分析可知，对于比较大的数据，编译器会把这些数据放到常量池中，在指令序列中则仅指明位置而已。

再看下面的代码：

```
public class HelloWorld03 {
    /**
```

```
     * @param args
     */
    public static void main(String[] args) {
        String stringvalue1 = "H";
        String stringvlaue2 = "H";
        String stringvalue3 = "Hello";
        char c = 'H';
    }
}
```

对应的指令序列为:

```
 0 ldc #16 <H>
 2 astore_1
 3 ldc #16 <H>
 5 astore_2
 6 ldc #18 <Hello>
 8 astore_3
 9 bipush 72
11 istore 4
13 return
```

通过上面例子可以看出，字符串也是放到常量池中的，而且相同的字符串在常量池中只有一份。字符则把它当成是一个 int 类型压入到操作数栈中。

请读者再看下面的代码:

```
public class HelloWorld05 {
    /**
     * @param args
     */
    public static void main(String[] args) {
        int result = 0;
        for(int i=0; i<10; i++){
            result = result + i;
        }
    }
}
```

对应的指令为:

```
 0 iconst_0
 1 istore_1
 2 iconst_0
 3 istore_2
 4 goto 14 (+10)
 7 iload_1
 8 iload_2
 9 iadd
10 istore_1
11 iinc 2 by 1
14 iload_2
15 bipush 10
17 if_icmplt 7 (-1
20 return
```

上述指令的具体说明如下：

- iconst_0：表示将 0 压入操作数栈；
- istore_1：表示将栈顶数据弹出存放到变量区，索引号为 1(即给 result 这个变量赋值)；
- iconst_0：表示将 0 压入操作数栈；
- istore_2：表示将栈顶数据弹出存放到变量区，索引号为 2(即给 i 这个变量赋值，int i=0)；
- goto 14：表示跳到 14 行去。在 14 行中，iload_2 表示把变量区索引号为 2 的值压入操作数栈(即取出 i 变量)；
- bipush 10：表示把 10 这个值压入操作数栈；
- if_icmplt 7：表示弹出操作数栈的两个数据，并判断操作数栈中的两个数的大小(即是否 i<10？)，如果是，则跑到第 7 行，否则，直接往下执行了。在第 7 行中，iload_1 即把变量区索引号为 1 的值压入操作数栈(即 result 变量)；
- iload_2：表示把变量区索引号为 2 的值压入操作数栈(即 i 变量)；
- iadd：把两个操作数栈中的数据弹出并相加，把结果重新压入操作数栈；
- istore_1：表示把操作数栈中的数据弹出，并存放到变量区索引号为 1 的地方(即更新了 result 变量的值)；
- iinc 2 by 1：表示的是将变量区索引号为 2 的值自增 1；
- 最后到了 14 行，将重复上述的过程。

常量值可以在字节码中跟随在操作码之后，例如，将 byte 和 short 类型常量压入栈的操作如表 7-4 所示。

表 7-4　将 byte 和 short 类型常量压入栈的操作说明

操 作 码	操 作 数	说　明
bipush	一个 byte 类型的数	将 byte 类型的数转换为 int 类型的数，然后压入栈
sipush	一个 short 类型的数	将 short 类型的数转换为 int 类型的数，然后压入栈

从常量池中取出常量的操作说明如表 7-5 所示。

表 7-5　从常量池中取出常量的操作说明

操 作 码	操 作 数	说　明
ldc	无符号 8 位数 indexbyte	从由 indexbyte 指向的常量池入口中取出一个字长的值，然后将其压入栈
ldc_w	无符号 16 位数 indexshort	从由 indexshort 指向的常量池入口中取出一个字长的值，然后将其压入栈
ldc2_w	无符号 16 位数 indexshort	从由 indexshort 指向的常量池入口中取出两个字长的值，然后将其压入栈

上述 3 个操作码是从常量池中取出常量，然后将其压入栈，这些操作码的操作码表示

常量池索引，Java 虚拟机通过给定的索引查找相应的常量池入口，决定这些常量的类型和值，并把它们压入栈。

常量池索引是一个无符号值，ldc 和 ldc_w 是把一个字长的项压入栈，区别是 ldc 的索引只有一个 8 位，只能指向常量池中 1～255 范围的位置。ldc_w 的索引有 16 位，可以指向 1～65535 范围的位置。

而通用栈操作的说明如表 7-6 所示。

表 7-6　通用栈操作的说明

操 作 码	操 作 数	说　明
nop	(无)	不做任何操作
pop	(无)	从操作数栈弹出栈顶部的一个字
pop2	(无)	从操作数栈弹出最顶端的两个字
swap	(无)	交换栈顶部的两个字
dup	(无)	复制栈顶部的一个字
dup2	(无)	复制栈顶部的两个字
dup_x1	(无)	复制栈顶部的一个字，并将复制内容及原来弹出的两个字长的内容压入栈
dup_x2	(无)	复制栈顶部的一个字，并将复制内容及原来弹出的三个字长的内容压入栈
dup2_x1	(无)	复制栈顶部的两个字，并将复制内容及原来弹出的三个字长的内容压入栈
dup2_x2	(无)	复制栈顶部的两个字，并将复制内容及原来弹出的四个字长的内容压入栈

把局部变量压入栈的操作说明如表 7-7 所示。

表 7-7　将一个字长的局部变量压入栈的操作说明

操 作 码	操 作 数	说　明
iload	vindex	将位置为 vindex 的 int 类型的局部变量压入栈
iload_0	(无)	将位置为 0 的 int 类型的局部变量压入栈
iload_1	(无)	将位置为 1 的 int 类型的局部变量压入栈
iload_2	(无)	将位置为 2 的 int 类型的局部变量压入栈
iload_3	(无)	将位置为 3 的 int 类型的局部变量压入栈
fload	vindex	将位置为 vindex 的 float 类型的局部变量压入栈
fload_0	(无)	将位置为 0 的 float 类型的局部变量压入栈
fload_1	(无)	将位置为 1 的 float 类型的局部变量压入栈
fload_2	(无)	将位置为 2 的 float 类型的局部变量压入栈
fload_3	(无)	将位置为 3 的 float 类型的局部变量压入栈

将两个字长的局部变量压入栈的操作说明如表 7-8 所示。

表 7-8 将两个字长的局部变量压入栈的操作说明

操 作 码	操 作 数	说 明
lload	vindex	将位置为 vindex 和(vindex+1)的 long 类型的局部变量压入栈
lload_0	(无)	将位置为 0 和 1 的 long 类型的局部变量压入栈
lload_1	(无)	将位置为 1 和 2 的 long 类型的局部变量压入栈
lload_2	(无)	将位置为 2 和 3 的 long 类型的局部变量压入栈
lload_3	(无)	将位置为 3 和 4 的 long 类型的局部变量压入栈
dload	vindex	将位置为 vindex 和(vindex+1)的 double 类型的局部变量压入栈
dload_0	(无)	将位置为 0 和 1 的 double 类型的局部变量压入栈
dload_1	(无)	将位置为 1 和 2 的 double 类型的局部变量压入栈
dload_2	(无)	将位置为 2 和 3 的 double 类型的局部变量压入栈
dload_3	(无)	将位置为 3 和 4 的 double 类型的局部变量压入栈

将对象引用局部变量压入栈的操作说明如表 7-9 所示。

表 7-9 将对象引用局部变量压入栈的操作说明

操 作 码	操 作 数	说 明
aload	vindex	将位置为 vindex 的对象引用局部变量压入栈
aload_0	(无)	将位置为 0 的对象引用局部变量压入栈
aload_1	(无)	将位置为 1 的对象引用局部变量压入栈
aload_2	(无)	将位置为 2 的对象引用局部变量压入栈
aload_3	(无)	将位置为 3 的对象引用局部变量压入栈

弹出栈顶元素，将其赋给局部变量的操作说明如表 7-10 所示。

表 7-10 赋值给局部变量的操作说明

操 作 码	操 作 数	说 明
istore	vindex	从栈中弹出 int 类型值，然后将其存到位置为 vindex 的局部变量中
istore_0	(无)	从栈中弹出 int 类型值，然后将其存到位置为 0 的局部变量中
istore_1	(无)	从栈中弹出 int 类型值，然后将其存到位置为 1 的局部变量中
istore_2	(无)	从栈中弹出 int 类型值，然后将其存到位置为 2 的局部变量中
istore_3	(无)	从栈中弹出 int 类型值，然后将其存到位置为 3 的局部变量中
fstore	vindex	从栈中弹出 float 类型值，然后将其存到位置为 vindex 的局部变量中
fstore_0	(无)	从栈中弹出 float 类型值，然后将其存到位置为 0 的局部变量中
fstore_1	(无)	从栈中弹出 float 类型值，然后将其存到位置为 1 的局部变量中
fstore_2	(无)	从栈中弹出 float 类型值，然后将其存到位置为 2 的局部变量中
fstore_3	(无)	从栈中弹出 float 类型值，然后将其存到位置为 3 的局部变量中

弹出对象引用，并将其赋值给局部变量的操作说明如表 7-11 所示。

表 7-11 赋值给局部变量的操作说明

操作码	操作数	说明
lstore	vindex	从栈中弹出 long 类型值，然后将其存到位置为 vindex 和(vindex+1)的局部变量中
lstore_0	(无)	从栈中弹出 long 类型值，然后将其存到位置为 0 和 1 的局部变量中
lstore_1	(无)	从栈中弹出 long 类型值，然后将其存到位置为 1 和 2 的局部变量中
lstore_2	(无)	从栈中弹出 long 类型值，然后将其存到位置为 2 和 3 的局部变量中
lstore_3	(无)	从栈中弹出 long 类型值，然后将其存到位置为 3 和 4 的局部变量中
dstore	vindex	从栈中弹出 double 类型值，然后将其存到位置为 vindex 和(vindex+1)的局部变量中
dstore_0	(无)	从栈中弹出 double 类型值，然后将其存到位置为 0 和 1 的局部变量中
dstore_1	(无)	从栈中弹出 double 类型值，然后将其存到位置为 1 和 2 的局部变量中
dstore_2	(无)	从栈中弹出 double 类型值，然后将其存到位置为 2 和 3 的局部变量中
dstore_3	(无)	从栈中弹出 double 类型值，然后将其存到位置为 3 和 4 的局部变量中
astore	vindex	从栈中弹出对象引用，然后将其存到位置为 vindex 的局部变量中
astore_0	(无)	从栈中弹出对象引用，然后将其存到位置为 0 的局部变量中
astore_1	(无)	从栈中弹出对象引用，然后将其存到位置为 1 的局部变量中
astore_2	(无)	从栈中弹出对象引用，然后将其存到位置为 2 的局部变量中
astore_3	(无)	从栈中弹出对象引用，然后将其存到位置为 3 的局部变量中
astore	vindex	从栈中弹出对象引用，然后将其存到位置为 vindex 的局部变量中

另外，通过无符号 8 位局部变量索引，可以把方法中局部变量的数量限制在 256 以下。一条单独的 wide 指令可以将 8 位的索引再扩展 8 位，这样就可以把局部变量数的限制扩展到 65536，如表 7-12 所示。

表 7-12 赋值给局部变量的操作说明

操作码	操作数	说明
wide	iload,index	从局部变量位置为 index 的地方取出 int 类型值，并将其压入栈
wide	lload ,index	从局部变量位置为 index 的地方取出 long 类型值，并将其压入栈
wide	fload,index	从局部变量位置为 index 的地方取出 float 类型值，并将其压入栈
wide	dload,index	从局部变量位置为 index 的地方取出 double 类型值，并将其压入栈
wide	aload,index	从局部变量位置为 index 的地方取出对象引用，并将其压入栈
wide	istore,index	从栈中弹出 int 类型值，将其存入位置为 index 的局部变量中
wide	lstore,index	从栈中弹出 long 类型值，将其存入位置为 index 的局部变量中
wide	fstore,index	从栈中弹出 float 类型值，将其存入位置为 index 的局部变量中
wide	dstore,index	从栈中弹出 double 类型值，将其存入位置为 index 的局部变量中
wide	astore,index	从栈中弹出对象引用，将其存入位置为 index 的局部变量中

跳转指令并不允许直接跳转到被 wide 指令修改过的操作码。

Java

第 8 章

内存异常和垃圾处理

对于 C 和 C++的开发人员来说，在内存管理领域应该能够游刃有余。在计算机系统中，内存负责维护每一个对象生命的开始到终结。本章将详细讲解 Java 虚拟机对内存进行管理的基本知识，介绍 Java 虚拟机内存的各个区域，讲解这些区域的作用、服务对象以及其中可能产生的问题，为虚拟机内存管理知识的学习打下基础。

8.1 Java的内存分配管理

Java 内存分配与管理是 Java 的核心技术之一，一般 Java 在内存分配时会涉及以下区域。

- 寄存器：我们在程序中无法控制。
- 栈：存放基本类型的数据和对象的引用，但对象本身不存放在栈中，而存放在堆中。
- 堆：存放用 new 产生的数据。
- 静态域：存放在对象中用 static 定义的静态成员。
- 常量池：存放常量。
- 非 RAM 存储：硬盘等永久存储空间。

8.1.1 内存分配中的栈和堆

1. 栈

在函数中定义的一些基本类型的变量数据，还有对象的引用变量都在函数的栈内存中分配。当在一段代码块中定义一个变量时，Java 就在栈中为这个变量分配内存空间，当该变量退出该作用域后，Java 会自动释放掉为该变量所分配的内存空间，该内存空间可以立即被另作他用。

栈也叫栈内存，是 Java 程序的运行区，是在线程创建时创建，它的生命期是跟随线程的生命期，线程结束栈内存也就释放，对于栈来说不存在垃圾回收问题，只要线程一结束，该栈就 Over。问题出来了：栈中存的是那些数据呢？又什么是格式呢？

栈中的数据都是以栈帧(Stack Frame)的格式存在，栈帧是一个内存区块，是一个数据集，是一个有关方法(Method)和运行期数据的数据集，当一个方法 A 被调用时就产生了一个栈帧 F1，并被压入到栈中，A 方法又调用了 B 方法，于是产生栈帧 F2 也被压入栈，执行完毕后，先弹出 F2 栈帧，再弹出 F1 栈帧，遵循“先进后出”原则。

那栈帧中到底存在着什么数据呢？在栈帧中主要保存如下三类数据：

- 本地变量(Local Variables)：包括输入参数和输出参数以及方法内的变量；
- 栈操作(Operand Stack)：记录出栈、入栈的操作；
- 栈帧数据(Frame Data)：包括类文件、方法等。

光说比较枯燥，我们画个图来理解一下 Java 栈，如图 8-1 所示。

在图 8-1 中，在一个栈中有两个栈帧，栈帧 2 是最先被调用的方法，先入栈，然后方法 2 又调用了方法 1，栈帧 1 处于栈顶的位置，栈帧 2 处于栈底，执行完毕后，依次弹出栈帧 1 和栈帧 2，线程结束，栈释放。

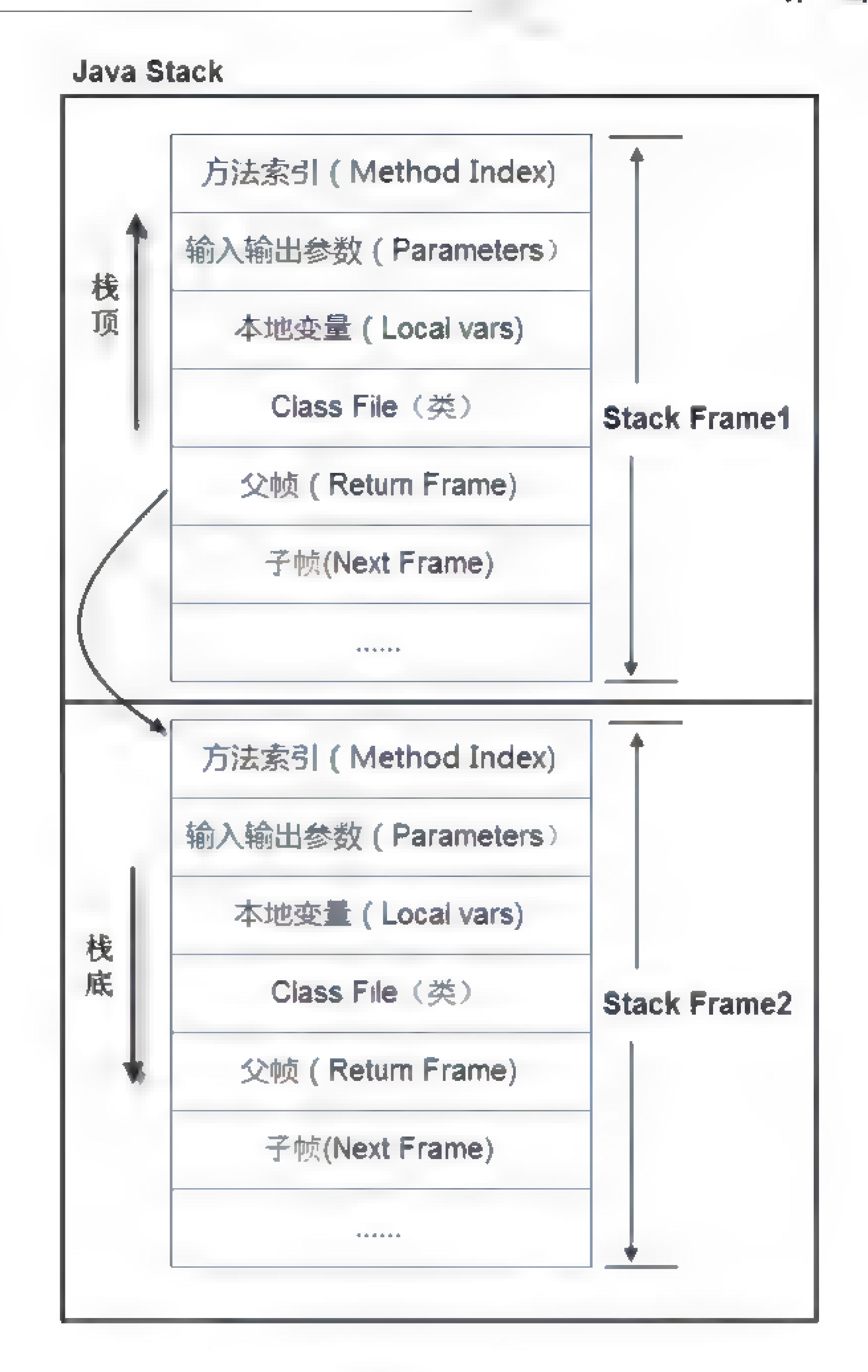


图 8-1 Java 栈

2. 堆

堆内存用来存放由关键字 new 创建的对象和数组。在堆中分配的内存，由 Java 虚拟机的自动垃圾回收器来管理。

在堆中产生了一个数组或对象后，还可以在栈中定义一个特殊的变量，让栈中这个变量的取值等于数组或对象在堆内存中的首地址，栈中的这个变量就成了数组或对象的引用变量。引用变量就相当于是为数组或对象起的一个名称，以后就可以在程序中使用栈中的引用变量来访问堆中的数组或对象。引用变量就相当于是为数组或者对象起的一个名称。

引用变量是普通的变量，定义时在栈中分配，引用变量在程序运行到其作用域之外后被释放。而数组和对象本身在堆中分配，即使程序运行到使用 new 产生数组或者对象的语句所在的代码块之外，数组和对象本身占据的内存不会被释放，数组和对象在没有引用变量指向它的时候，才变为垃圾，不能再被使用，但仍然占据内存空间不放，在随后的一

个不确定的时间被垃圾回收器收走(释放掉)。这也是 Java 比较占内存的原因。

实际上，栈中的变量指向堆内存中的变量，这就是 Java 中的指针。

3．常量池(Constant Pool)

常量池指的是在编译期被确定，并被保存在已编译的.class 文件中的一些数据。除了包含代码中所定义的各种基本类型(如 int、long 等)和对象型(如 String 及数组)的常量值(Final)还包含一些以文本形式出现的符号引用，比如：

- 类和接口的全限定名。
- 字段的名称和描述符。
- 方法和名称和描述符。

虚拟机必须为每个被装载的类型维护一个常量池。常量池就是该类型所用到常量的一个有序集和，包括直接常量(string、integer 和 floating point 常量)和对其他类型，字段和方法的符号引用。

对于 String 常量，它的值是在常量池中的。而 JVM 中的常量池在内存当中是以表的形式存在的，对于 String 类型，有一张固定长度的 CONSTANT_String_info 表用来存储文字字符串值，但是该表只存储文字字符串值，并不存储符号引用。在程序执行的时候，常量池会储存在 Method Area(方法区域)中，而不是堆中。

一个 JVM 实例只存在一个堆内存，堆内存的大小是可以调节的。类加载器读取了类文件后，需要把类、方法、常变量放到堆内存中，以方便执行器执行，堆内存分为以下三部分。

1) Permanent Space 永久存储区

永久存储区是一个常驻内存区域，用于存放 JDK 自身所携带的 Class Interface 的元数据。也就是说，它存储的是运行环境必需的类信息，被装载进此区域的数据是不会被垃圾回收器回收掉的，关闭 JVM 才会释放此区域所占用的内存。

2) Young Generation Space 新生区

新生区是类的诞生、成长、消亡的区域，一个类在这里产生、应用，最后被垃圾回收器收集，结束生命。新生区又分为两部分：伊甸区(Eden Space)和幸存者区(Survivor Space)，所有的类都是在伊甸区被 new(新建)出来的。幸存区有两个：0 区(Survivor 0 space)和 1 区(Survivor 1 space)。当伊甸园的空间用完时，程序又需要创建对象，JVM 的垃圾回收器将对伊甸园区进行垃圾回收，将伊甸园区中的不再被其他对象所引用的对象进行销毁。然后将伊甸园中的剩余对象移动到幸存 0 区。若幸存 0 区也满了，再对该区进行垃圾回收，然后移动到 1 区。那如果 1 区也满了呢？再移动到养老区。

3) Tenure generation space 养老区

养老区用于保存从新生区筛选出来的 Java 对象，一般池对象都在这个区域活跃。

上述三个区的示意图如图 8-2 所示。





图 8-2　堆内存的三个区

注意：为什么要把 JVM 堆和 JVM 栈区分出来呢?JVM 栈中不是也可以存储数据吗?

(1) 从软件设计的角度看，JVM 栈代表了处理逻辑，而 JVM 堆代表了数据。这样分开，使得处理逻辑更为清晰。分而治之的思想。这种隔离、模块化的思想在软件设计的方方面面都有体现。

(2) JVM 堆与 JVM 栈的分离，使得 JVM 堆中的内容可以被多个 JVM 栈共享(也可以理解为多个线程访问同一个对象)。这种共享的收益是很多的。一方面这种共享提供了一种有效的数据交互方式(如共享内存)，另一方面，JVM 堆中的共享常量和缓存可以被所有 JVM 栈访问，节省了空间。

(3) JVM 栈因为运行时的需要，比如保存系统运行的上下文，需要进行地址段的划分。由于 JVM 栈只能向上增长，因此就会限制住 JVM 栈存储内容的能力。而 JVM 堆不同，JVM 堆中的对象是可以根据需要动态增长的，因此 JVM 栈和 JVM 堆的拆分，使得动态增长成为可能，相应 JVM 栈中只需记录 JVM 堆中的一个地址即可。

(4) 面向对象就是 JVM 堆和 JVM 栈的完美结合。其实，面向对象方式的程序与以前结构化的程序在执行上没有任何区别。但是，面向对象的引入，使得对待问题的思考方式发生了改变，而更接近于自然方式的思考。当我们把对象拆开，你会发现，对象的属性其实就是数据，存放在 JVM 堆中；而对象的行为(方法)，就是运行逻辑，放在 JVM 栈中。我们在编写对象的时候，其实即编写了数据结构，也编写的处理数据的逻辑。不得不承认，面向对象的设计，确实很美。

8.1.2　堆和栈的合作

Java 的堆是一个运行时数据区，类的对象从中分配空间。这些对象通过 new、newarray、anewarray 和 multianewarray 等指令建立，它们不需要程序代码来显式地释放。堆是由垃圾回收来负责的，堆的优势是可以动态地分配内存大小，生存期也不必事先告诉编译器，因为它是在运行时动态分配内存的，Java 的垃圾收集器会自动收走这些不再使用的数据。但缺点是，由于要在运行时动态分配内存，存取速度较慢。

栈的优势是存取速度比堆要快，仅次于寄存器，栈数据可以共享。但缺点是，存在栈

中的数据大小与生存期必须是确定的，缺乏灵活性。栈中主要存放一些基本类型的变量数据(int、short、long、byte、float、double、boolean、char)和对象句柄(引用)。

栈有一个很重要的特殊性，就是存在栈中的数据可以共享。假设我们同时定义：

```
int a = 3;
int b = 3;
```

编译器先处理 int a = 3；首先它会在栈中创建一个变量为 a 的引用，然后查找栈中是否有 3 这个值，如果没找到，就将 3 存放进来，然后将 a 指向 3。接着处理 int b = 3；在创建完 b 的引用变量后，因为在栈中已经有 3 这个值，便将 b 直接指向 3。这样，就出现了 a 与 b 同时均指向 3 的情况。如果这时再令 a=4；那么编译器会重新搜索栈中是否有 4 值，如果没有，则将 4 存放进来，并令 a 指向 4；如果已经有了，则直接将 a 指向这个地址。因此 a 值的改变不会影响 到 b 的值。

要注意这种数据的共享与两个对象的引用同时指向一个对象的这种共享是不同的，因为这种情况 a 的修改并不会影响到 b，它是由编译器完成的，有利于节省空间。而一个对象引用变量修改了这个对象的内部状态，会影响到另一个对象引用变量。

在 Java 中，String 是一个特殊的包装类数据。可以用如下两种的形式来创建。

```
String str = new String("abc");
String str = "abc";
```

其中第一种是用 new()来新建对象的，它会在存放于堆中。每调用一次就会创建一个新的对象。而第二种是先在栈中创建一个对 String 类的对象引用变量 str，然后通过符号引用去字符串常量池 里找有没有"abc"，如果没有，则将"abc"存放进字符串常量池 ，并令 str 指向"abc"，如果已经有"abc"则直接令 str 指向"abc"。

比较类里面的数值是否相等时使用 equals()方法；当测试两个包装类的引用是否指向同一个对象时，用= =，下面用例子说明上面的理论。

```
String str1 = "abc";
String str2 = "abc";
System.out.println(str1==str2); //true
```

由此可以看出 str1 和 str2 是指向同一个对象的。

```
String str1 =new String ("abc");
String str2 =new String ("abc");
System.out.println(str1==str2); // false
```

用 new 的方式的功能是生成不同的对象，每一次生成一个。因此用第二种方式创建多个"abc"字符串，在内存中其实只存在一个对象而已。这种写法有利于节省内存空间，同时它可以在一定程度上提高程序的运行速度，因为 JVM 会自动根据栈中数据的实际情况来决定是否有必要创建新对象。而对于代码“String str = new String("abc");”，则一概在堆中创建新对象，而不管其字符串值是否相等，是否有必要创建新对象，从而加重了程序的负担。

另一方面，要注意在使用诸如“String str = "abc";”的格式定义类时，总是想当然地认为，创建了 String 类的对象 str。此时会担心对象可能并没有被创建，而可能只是指向一个

先前已经创建的对象。只有通过方法 new()才能保证每次都创建一个新的对象。

由于 String 类的 immutable 性质，当 String 变量需要经常变换其值时，应该考虑使用 StringBuffer 类以提高程序效率。因为 String 不属于 8 种基本数据类型，String 是一个对象。所以对象的默认值是 null，所以 String 的默认值也是 null；但它又是一种特殊的对象，有其他对象没有的一些特性。由此可见，new String()和 new String("")都是申明一个新的空字符串，是空串不是 null。

请看下面的代码：

```
String s0="kvill";
String s1="kvill";
String s2="kv" + "ill";
System.out.println( s0==s1 );
System.out.println( s0==s2 );
```

运行结果为：

```
true true
```

Java 会确保一个字符串常量只有一个拷贝。因为上述例子中的 s0 和 s1 中的"kvill"都是字符串常量，它们在编译期就被确定了，所以 s0==s1 为 true；而"kv"和"ill"也都是字符串常量，当一个字符串由多个字符串常量连接而成时，它自己肯定也是字符串常量，所以 s2 也同样在编译期就被解析为一个字符串常量，所以 s2 也是常量池中"kvill"的一个引用。所以我们得出 s0==s1==s2；用 new String() 创建的字符串不是常量，不能在编译期就确定，所以 new String() 创建的字符串不放入常量池中，它们有自己的地址空间。

再看下面的代码：

```
String s0="kvill";
String s1=new String("kvill");
String s2="kv" + new String("ill");
System.out.println( s0==s1 );
System.out.println( s0==s2 );
System.out.println( s1==s2 );
```

运行结果为：

```
false false false
```

在上述代码中，s0 还是常量池中"kvill"的应用，s1 因为无法在编译期确定，所以是运行时创建的新对象"kvill"的引用，s2 因为有后半部分 new String("ill")所以也无法在编译期确定，所以也是一个新创建对象"kvill"的应用，明白了这些也就知道为何得出此结果了。

另外，存在于.class 文件中的常量池，在运行时被 JVM 装载，并且可以扩充。String 的 intern()方法就是扩充常量池的一个方法；当一个 String 实例 str 调用 intern()方法时，Java 查找常量池中是否有相同 Unicode 的字符串常量，如果有则返回其的引用，如果没有则在常量池中增加一个 Unicode 等于 str 的字符串并返回它的引用。请看下面的演示示例：

```
String s0= "kvill";
String s1=new String("kvill");
String s2=new String("kvill");
System.out.println( s0==s1 );
```

```
System.out.println( "**********" );
s1.intern();
s2=s2.intern(); //把常量池中"kvill"的引用赋给 s2
System.out.println( s0==s1);
System.out.println( s0==s1.intern() );
System.out.println( s0==s2 );
```

运行结果为：

```
false false //虽然执行了 s1.intern(),但它的返回值没有赋给 s1
true //说明 s1.intern()返回的是常量池中"kvill"的引用 true
```

另外，很多人认为使用 String.intern()方法可以将一个 String 类的保存到一个全局 String 表中，如果具有相同值的 Unicode 字符串已经在这个表中，那么该方法返回表中已有字符串的地址，如果在表中没有相同值的字符串，则将自己的地址注册到表中"如果我把他说的这个全局的 String 表理解为常量池的话，如果在表中没有相同值的字符串，则将自己的地址注册到表中"是错的。请看下面的演示示例：

```
String s1=new String("kvill");
String s2=s1.intern();
System.out.println( s1==s1.intern() );
System.out.println( s1+" "+s2 );
System.out.println( s2==s1.intern() );
```

运行结果为：

```
false kvill kvill true
```

在这个类中我们没有声名一个"kvill"常量，所以常量池中一开始是没有"kvill"的，当我们调用 s1.intern()后就在常量池中新添加了一个"kvill"常量，原来的不在常量池中的"kvill"仍然存在，也就不是"将自己的地址注册到常量池中"了。

s1= =s1.intern()为 false 说明原来的"kvill"仍然存在；s2 现在为常量池中"kvill"的地址，所以有 s2= =s1.intern()为 true。

通过使用 equals()，String 可以比较两字符串的 Unicode 序列是否相当，如果相等返回 true。而"= ="是比较两字符串的地址是否相同，也就是是否是同一个字符串的引用。

String 的实例一旦生成就不会再改变了，比如：

```
String str="kv"+"ill"+" "+"ans";
```

上述 str 有 4 个字符串常量，首先"kv"和"ill"生成了"kvill"存在内存中，然后"kvill"又和" "生成"kvill"存在内存中，最后又和生成了"kvill ans"。并把这个字符串的地址赋给了 str，就是因为 String 的"不可变"产生了很多临时变量，这也就是为什么建议用 StringBuffer 的原因了，因为 StringBuffer 是可改变的。

Java 内存分配与管理是 Java 的核心技术之一，之前我们曾介绍过 Java 的内存管理与内存泄露以及 Java 垃圾回收方面的知识，今天我们再次深入 Java 核心，详细介绍一下 Java 在内存分配方面的知识。

8.2　运行时的数据区域

Java 通过自身的动态内存分配和垃圾回收机制，可以使 Java 程序员不用像 C++程序员那么头疼内存的分配与回收。对于这一点来说，相信熟悉 COM 机制的朋友对于引用计数管理内存的方式深有感触。通过 Java 虚拟机的自动内存管理机制，不仅降低了编码的难度，而且不容易出现内存泄露和内存溢出的问题。但是这过于理想的愿望正是由于把内存的控制权交给了 Java 虚拟机，一旦出现内存泄露和溢出，我们就必须翻过 Java 虚拟机自动内存管理这堵高墙去排查错误。所以本节将详细讲解 JVM 运行时数据区域的划分、作用以及可能出现的异常。

根据《Java 虚拟机规范》的规定，Java 虚拟机在执行 Java 程序时，即运行时环境下会把其所管理的内存划分为几个不同的数据区域。有的区域伴随虚拟机进程的启动而创建，死亡而销毁；有些区域则是依赖用户线程的启动时创建，结束时销毁。所有线程共享方法区和堆，虚拟机栈、本地方法栈和程序计数器是线程隔离的数据区。Java 虚拟机运行时的数据区结构如图 8-3 所示。

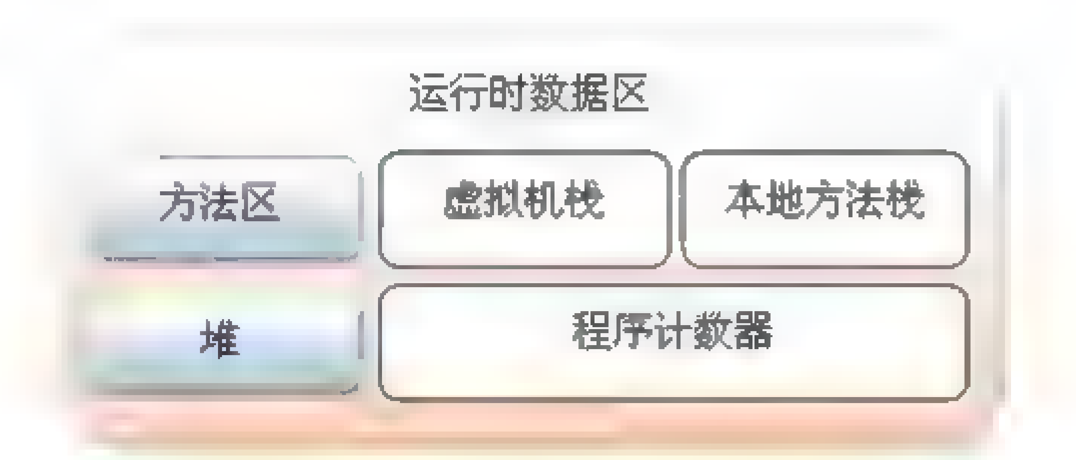


图 8-3　Java 虚拟机运行时的数据区结构

Java 虚拟机内存模型中定义的访问操作与物理计算机处理的基本一致。Java 通过多线程机制使得多个任务同时执行处理，所有的线程共享 JVM 内存区域 Main Memory，而每个线程又单独的有自己的工作内存，当线程与内存区域进行交互时，数据从主存拷贝到工作内存，进而交由线程处理(操作码+操作数)。

8.2.1　程序计数器(Program Counter Register)

程序计数器是一块较小的内存空间，其作用相当于当前线程所执行的字节码的行号指示器。在虚拟机的概念模型里，字节码解释器工作时通过改变这个计数器的值来选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成。

由于 Java 虚拟机的多线程是通过线程轮流切换并分配处理器执行时间的方式来实现的，在任何一个确定的时刻，一个处理器(对于多核处理器来说是一个内核)只会执行一条线程中的指令。因此为了线程切换后能恢复到正确的执行位置，每条线程都需要有一个独立的程序计数器，各条线程之间的计数器互不影响，独立存储，我们称这类内存区域为

"线程私有"的内存。

如果线程正在执行的是一个 Java 方法，这个计数器记录的是正在执行的虚拟机字节码指令的地址。如果正在执行的是 Native 方法，这个计数器值则为空(Undefined)。此内存区域是唯一一个在 Java 虚拟机规范中没有规定任何 OutOfMemoryError 情况的区域。

因为操作系统通过时间片轮流的多线程并发方式，任何时刻处理器只会处理当前线程的指令。线程间切换的并发要求每个线程都需要有一个私有的程序计数器，程序计数器间互不影响。

程序计数器存储当前线程下一条要执行的字节码的地址，占用内存空间较小。所有的控制执行流程，分支、循环、返回、异常等功能都在程序计数器的指示范围之内，字节码解释器通过改变程序计数器的值来获取下一条要执行的字节码的指令。

8.2.2 Java 的虚拟机栈 VM Stack

虚拟机栈是类中的方法的执行过程的内存模型。与程序计数器一样，Java 虚拟机栈(Java Virtual Machine Stacks)也是线程私有的，它的生命周期与线程相同。虚拟机栈描述的是 Java 方法执行的内存模型：每个方法被执行的时候都会同时创建一个栈帧(Stack Frame)用于存储局部变量表、操作数栈、动态链接、方法出口等信息。每一个方法被调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。

对于方法调用来说，很有必要了解下栈帧的概念。

虚拟机在执行每个方法的调用时会创建一个栈帧的数据结构，它是虚拟机运行时数据区中的虚拟机栈的栈元素。每个方法的调用过程，就对应着一个栈帧在虚拟机里的入栈出栈的过程。栈帧包括了方法的局部变量表、操作数栈、动态链接和方法出口等一些额外的附加信息。对于活动线程中栈顶的帧栈，称为当前栈帧，这个栈帧所关联的方法称为当前方法，正在执行的字节码指令都只针对当前有效栈帧进行操作。

在栈帧的基础上，不难理解虚拟机栈的内存结构。Java 虚拟机规范规定虚拟机栈的大小是可以固定的或者动态分配大小。Java 虚拟机实现可以向程序员提供对 Java 栈的初始大小的控制，以及在动态扩展或者收缩 Java 栈的情况下，控制 Java 栈的最大值和最小值。

下面列出的异常情况与 Java 栈相关。

- 如果线程请求的栈深度大于虚拟机所允许的深度，则 Java 虚拟机将抛出 StackOverflowError 异常。
- 如果虚拟机栈可以动态扩展，但是无法申请到足够的内存来实现扩展，或者不能得到足够的内存为一个新线程创建初始 Java 栈，则 Java 虚拟机将抛出 OutOfMemoryError 异常。

有人通常把 Java 内存区分为堆内存(Heap)和栈内存(Stack)，其实 Java 内存区域的划分实际上远比这复杂。这种划分方式的流行只能说明大多数程序员最关注的、与对象内存分配关系最密切的内存区域是这两块。其中所指的"栈"就是现在讲的虚拟机栈，或者说是虚拟机栈中的局部变量表部分。

在局部变量表中存放了编译期可知的各种基本数据类型(boolean、byte、char、short、int、float、long、double)、对象引用(reference 类型，它不等同于对象本身，根据不同的虚

拟机实现，它可能是一个指向对象起始地址的引用指针，也可能指向一个代表对象的句柄或者其他与此对象相关的位置)和 returnAddress 类型(指向了一条字节码指令的地址)。

其中 64 位长度的 long 和 double 类型的数据会占用两个局部变量空间(Slot)，其余的数据类型只占用一个。局部变量表所需的内存空间在编译期间完成分配，当进入一个方法时，这个方法需要在帧中分配多大的局部变量空间是完全确定的，在方法运行期间不会改变局部变量表的大小。

在 Java 虚拟机规范中，对这个区域规定了两种异常状况：如果线程请求的栈深度大于虚拟机所允许的深度，将抛出 StackOverflowError 异常；如果虚拟机栈可以动态扩展(当前大部分的 Java 虚拟机都可动态扩展，只不过 Java 虚拟机规范中也允许固定长度的虚拟机栈)，当扩展时无法申请到足够的内存时会抛出 OutOfMemoryError 异常。

8.2.3　本地方法栈 Native Method Stack

在本地方法栈中执行的是非 Java 语言编写的代码，例如 C 或 C++。虚拟机栈执行的是 Java 方法字节码服务，这是两者最大的区别。本地方法栈的是虚拟机使用本地方法服务的，如果提供本地方法栈，则它们通常在每个线程被创建时分配在每个线程基础上的。虚拟机规范中对本地方法栈中的方法使用的语言、使用方式与数据结构并没有强制规定，因此具体的虚拟机可以自由实现它。甚至有的虚拟机(譬如 Sun HotSpot 虚拟机)直接就把本地方法栈和虚拟机栈合二为一。

同虚拟机栈一样，本地方法栈也会出现与虚拟机栈类似的异常，也会抛出 StackOverflowError 和 OutOfMemoryError 异常。

8.2.4　Java 堆 Java Heap

Java 堆是类实例和数组的分配空间，是一块所有线程共享的内存区域。堆在虚拟机启动时创建，是 Java 虚拟机所管理的内存中最大一块。内存泄露和溢出的问题大都出现在堆区域，由此可见，对于大多数应用来说，Java 堆(Java Heap)是 Java 虚拟机所管理的内存中最大的一块。

从内存回收的角度看，由于现在收集器基本上都是采用的分代收集算法，Java 堆还可细分为新生代和老年代；从内存分配的角度看，线程共享的 Java 堆中可能划分出多个线程私有的分配缓存区。这种进一步的内存划分方式目的是更好地回收内存，或者更快地分配内存。

Java 堆是被所有线程共享的一块内存区域，在虚拟机启动时创建。此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例都在这里分配内存。这一点在 Java 虚拟机规范中的描述是：所有的对象实例以及数组都要在堆上分配，但是随着 JIT 编译器的发展与逃逸分析技术的逐渐成熟，栈上分配、标量替换优化技术将会导致一些微妙的变化发生，所有的对象都分配在堆上也渐渐变得不是那么“肯定”了。

Java 虚拟机规范规定堆在内存单元中只要在逻辑上是连续的即可，Java 堆是可以是固定大小的，或者按照需求做动态扩展，并且可以在一个大的堆变的不必要时收缩。Java 虚拟机的实现向程序员或者用户提供了对堆初始化大小的控制，以及对堆动态扩展和收缩的

最大值和最小值的控制。

下面的异常情况与 Java 堆相关：如果堆中没有可用内存完成类实例或者数组的分配，在对象数量达到最大堆的容量限制后将抛出 OutOfMemoryError 异常。

Java 堆是垃圾收集器管理的主要区域，因此很多时候也被称做“GC 堆”(Garbage Collected Heap)。如果从内存回收的角度看，由于现在收集器基本都是采用的分代收集算法，所以 Java 堆中还可以继续细分为：新生代和老年代。如果再细致一点，可以分为 Eden 空间、From Survivor 空间、To Survivor 空间等。如果从内存分配的角度看，线程共享的 Java 堆中可能划分出多个线程私有的分配缓冲区(Thread Local Allocation Buffer，TLAB)。但是无论如何划分，都与存放内容无关，无论哪个区域，存储的都仍然是对象实例。进一步划分的目的是为了更好地回收内存，或者更快地分配内存。

根据 Java 虚拟机规范的规定，Java 堆可以处于物理上不连续的内存空间中，只要逻辑上是连续的即可，就像我们的磁盘空间一样。在实现时，既可以实现成固定大小的，也可以是可扩展的，不过当前主流的虚拟机都是按照可扩展来实现的(通过-Xmx 和-Xms 控制)。

8.2.5 方法区 Method Area

方法区在虚拟机启动时创建，也是一块所有线程共享的内存区域。方法区用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。用一句话说就是方法区类似于传统语言的编译后代码的存储区。

方法区与 Java 堆一样，是各个线程共享的内存区域，它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。虽然 Java 虚拟机规范把方法区描述为堆的一个逻辑部分，但是它却有一个别名叫做 Non-Heap(非堆)，目的应该是与 Java 堆区分开来。

对于习惯在 HotSpot 虚拟机上开发和部署程序的开发者来说，很多人愿意把方法区称为“永久代”(Permanent Generation)，本质上两者并不等价，仅仅是因为 HotSpot 虚拟机的设计团队选择把 GC 分代收集扩展至方法区，或者说使用永久代来实现方法区而已。对于其他虚拟机(如 BEA JRockit、IBM J9 等)来说是不存在永久代的概念的。即使是 HotSpot 虚拟机本身，根据官方发布的路线图信息，现在也有放弃永久代并“搬家”至 Native Memory 来实现方法区的规划了。

Java 虚拟机规范对这个区域的限制非常宽松，除了和 Java 堆一样不需要连续的内存和可以选择固定大小或者可扩展外，还可以选择不实现垃圾收集。相对而言，垃圾收集行为在这个区域是比较少出现的，但并非数据进入了方法区就如永久代的名字一样“永久”存在了。这个区域的内存回收目标主要是针对常量池的回收和对类型的卸载，一般来说这个区域的回收“成绩”比较难以令人满意，尤其是类型的卸载，条件相当苛刻，但是这部分区域的回收确实是有必要的。在 Sun 公司的 BUG 列表中，曾出现过的若干个严重的 BUG 就是由于低版本的 HotSpot 虚拟机对此区域未完全回收而导致内存泄露。

虽然 Java 虚拟机规范在逻辑上把方法区描述为堆的一个部分，但是在垃圾回收方面的限制却比较宽松，宽松到方法区可以不用实现垃圾回收。但是，垃圾回收在方法区还是必

须有的，只是回收效果不是很明显。这个区域的回收目标主要针对的是常量池的回收和对类型的卸载。

方法区的大小也可以控制，以下异常与方法区相关：

如果方法区无法满足内存分配需求时，将会抛出 OutOfMemoryError 异常。

8.2.6　运行时常量池 Runtime Constant Pool

运行时常量池(Runtime Constant Pool)是方法区的一部分。在 Class 文件中除了有类的版本、字段、方法、接口等描述等信息外，还有一项信息是常量池(Constant Pool Table)，用于存放编译期生成的各种字面量和符号引用，这部分内容将在类加载后存放到方法区的运行时常量池中。

常量池是每个类的 Class 文件中存储编译期生成的各种字面量和符号引用的运行期表示，其数据结构是一种由无符号数和表组长的类似于 C 语言结构体的伪结构。另外，常量池也是方法区的一部分，类的常量池在该类的 Java class 文件被 Java 虚拟机成功地装载时创建，这部分内容在类加载后存放到方法区的运行时常量池中。

Java 虚拟机对 Class 文件的每一部分(自然也包括常量池)的格式都有严格的规定，每一个字节用于存储哪种数据都必须符合规范上的要求，这样才会被虚拟机认可、装载和执行。但对于运行时常量池来说，Java 虚拟机规范没有做任何细节的要求，不同的提供商实现的虚拟机可以按照自己的需要来实现这个内存区域。不过，一般来说，除了保存 Class 文件中描述的符号引用外，还会把翻译出来的直接引用也存储在运行时常量池中。

运行时常量池相对于 Class 文件常量池的另外一个重要特征是具备动态性，Java 语言并不要求常量一定只能在编译期产生，也就是并非预置入 Class 文件中常量池的内容才能进入方法区运行时常量池，运行期间也可能将新的常量放入池中，这种特性被开发人员利用得比较多的便是 String 类的 intern()方法。

既然运行时常量池是方法区的一部分，自然会受到方法区内存的限制，当常量池无法再申请到内存时会抛出 OutOfMemoryError 异常。运行时常量池属于方法区，自然也受到方法区内存大小的限制，以下异常与常量池有关：

在装载 class 文件时，如果常量池的创建需要比 Java 虚拟机的方法区中需求更多的内存时，将会抛出 OutOfMemoryError 异常。

注意：对于虚拟机运行时数据区域的划分及每个区域作用，存储内容及可能出现的异常有了一个大致的了解。Java 的自动内存分配和垃圾回收筑起的这道高墙，在出现内存泄露或者溢出的情况下，这道高墙就必须翻越了。

8.2.7　直接内存(Direct Memory)

直接内存并不是虚拟机运行时数据区的一部分，也不是 Java 虚拟机规范中定义的内存区域，但是这部分内存也被频繁地使用，而且也可能导致 OutOfMemoryError 异常出现，所以我们放到这里一起讲解。

从 JDK 1.4 版本开始，新加入了 NIO(New Input/Output)类，并且引入了一种基于通道(Channel)与缓冲区(Buffer)的 I/O 方式，它可以使用 Native 函数库直接分配堆外内存，然

后通过一个存储在 Java 堆里面的 DirectByteBuffer 对象作为这块内存的引用进行操作。这样能在一些场景中显著提高性能，因为避免了在 Java 堆和 Native 堆中来回复制数据。

显然，本机直接内存的分配不会受到 Java 堆大小的限制，但是既然是内存，肯定还是会受到本机总内存(包括 RAM 及 SWAP 区或者分页文件)的大小及处理器寻址空间的限制。服务器管理员配置虚拟机参数时，一般会根据实际内存设置“Xmx”等参数信息，但经常会忽略掉直接内存，使得各个内存区域的总和大于物理内存限制(包括物理上的和操作系统级的限制)，从而导致动态扩展时出现 OutOfMemoryError 异常。

8.3 对象访问

JVM 的逻辑内存模型如图 8-4 所示。

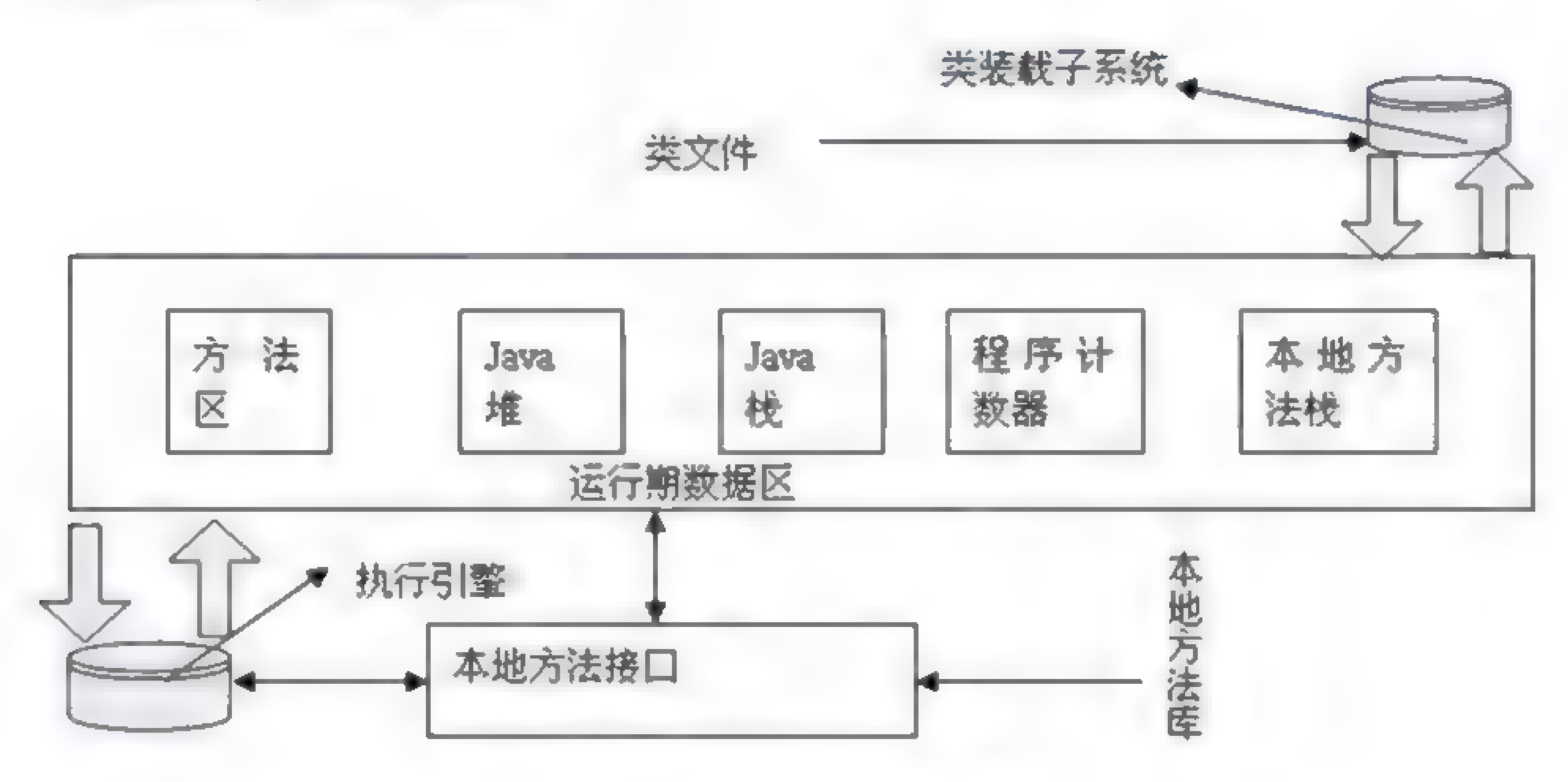


图 8-4　JVM 的逻辑内存模型

在本节的内容中，将通过逻辑内存模型来讲解对象访问的应用知识。

8.3.1　对象访问基础

当我们建立一个对象的时候是怎么进行访问的呢？在 Java 语言中，对象访问是如何进行的？对象访问在 Java 语言中无处不在，是最普通的程序行为，但即使是最简单的访问，也会却涉及 Java 栈、Java 堆、方法区这三个最重要内存区域之间的关联关系，如下面的代码：

```
Object obj = new Object();
```

假设这句代码出现在方法体中，那“Object obj”这部分的语义将会反映到 Java 栈的本地变量表中，作为一个 reference 类型数据出现。而“new Object()”这部分的语义将会反映到 Java 堆中，形成一块存储了 Object 类型所有实例数据值(Instance Data，对象中各个实例字段的数据)的结构化内存，根据具体类型以及虚拟机实现的对象内存布局(Object

Memory Layout)的不同，这块内存的长度是不固定的。另外，在 Java 堆中还必须包含能查找到此对象类型数据(如对象类型、父类、实现的接口、方法等)的地址信息，这些类型数据则存储在方法区中。

由于 reference 类型在 Java 虚拟机规范里面只规定了一个指向对象的引用，并没有定义这个引用应该通过哪种方式去定位，以及访问到 Java 堆中对象的具体位置，因此不同虚拟机实现的对象访问方式会有所不同，主流的访问方式有两种：使用句柄和直接指针。

(1) 如果使用句柄访问方式，Java 堆中将会划分出一块内存来作为句柄池，reference 中存储的就是对象的句柄地址，而句柄中包含了对象实例数据和类型数据各自的具体地址信息，如图 8-5 所示。

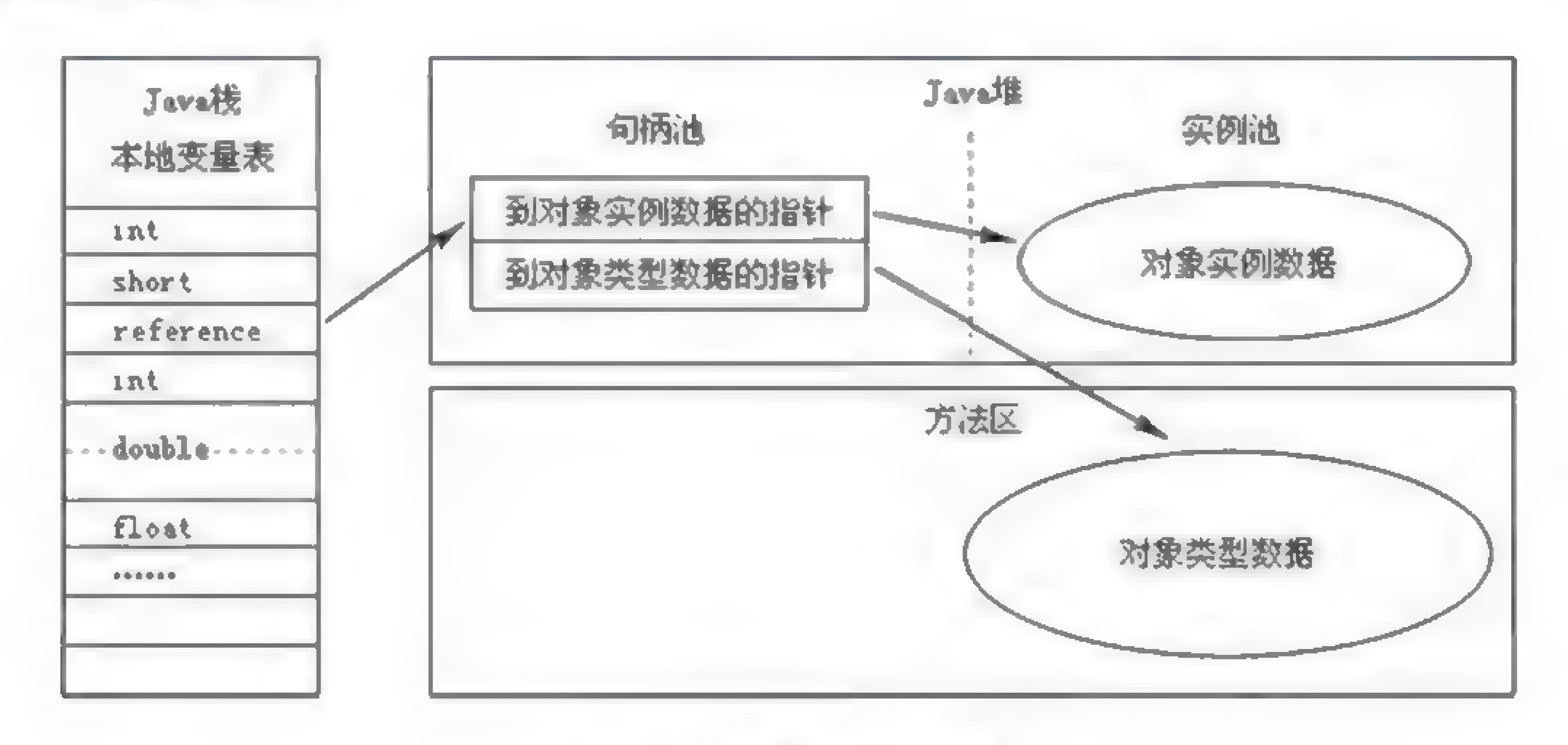


图 8-5　用句柄访问对象

(2) 如果使用直接指针访问方式，Java 堆对象的布局中就必须考虑如何放置访问类型数据的相关信息，reference 中直接存储的就是对象地址，如图 8-6 所示。

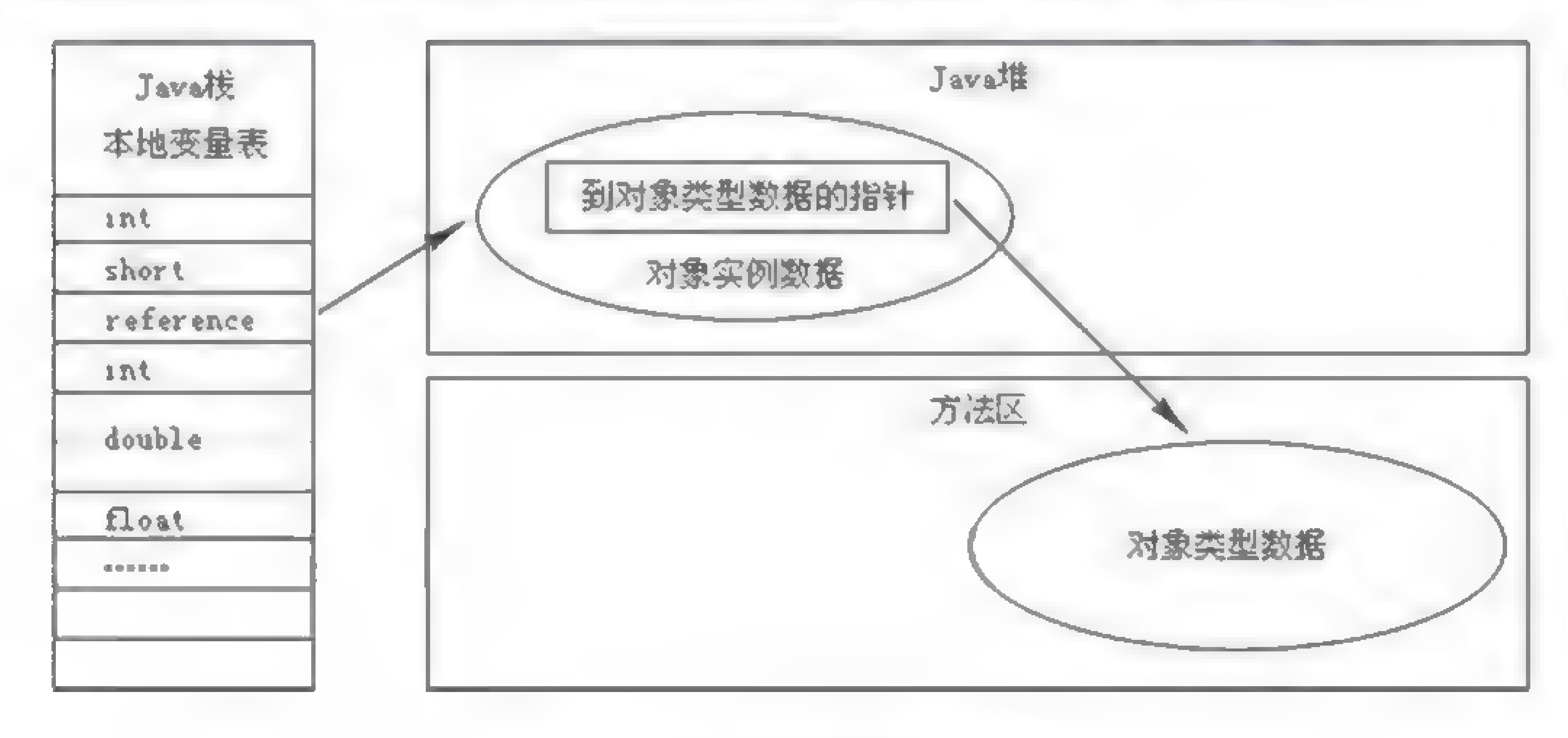


图 8-6　通过指针访问对象

这两种对象的访问方式各有优势，使用句柄访问方式的最大好处就是 reference 中存储的是稳定的句柄地址，在对象被移动(垃圾收集时移动对象是非常普遍的行为)时只会改变句柄中的实例数据指针，而 reference 本身不需要被修改。

使用直接指针访问方式的最大好处就是速度更快，它节省了一次指针定位的时间开销，由于对象的访问在 Java 中非常频繁，因此这类开销积少成多后也是一项非常可观的执行成本。就本书讨论的主要虚拟机 Sun HotSpot 而言，它是使用第二种方式进行对象访问的，但从整个软件开发的范围来看，各种语言和框架使用句柄来访问的情况也十分常见。

8.3.2 具体测试

在本节的内容中，将通过几个示例来演示 JVM 内存操作的基本知识。

1. Java 堆溢出

在示例中我们限制 Java 堆的大小为 20MB，不可扩展(将堆的最小值-Xms 参数与最大值-Xmx 参数设置为一样即可避免堆自动扩展)，通过参数-XX:+HeapDumpOnOutOfMemoryError 可以让虚拟机在出现内存溢出异常时 Dump 出当前的内存堆转储快照以便事后进行分析。具体参数设置分别如图 8-7～图 8-9 所示。

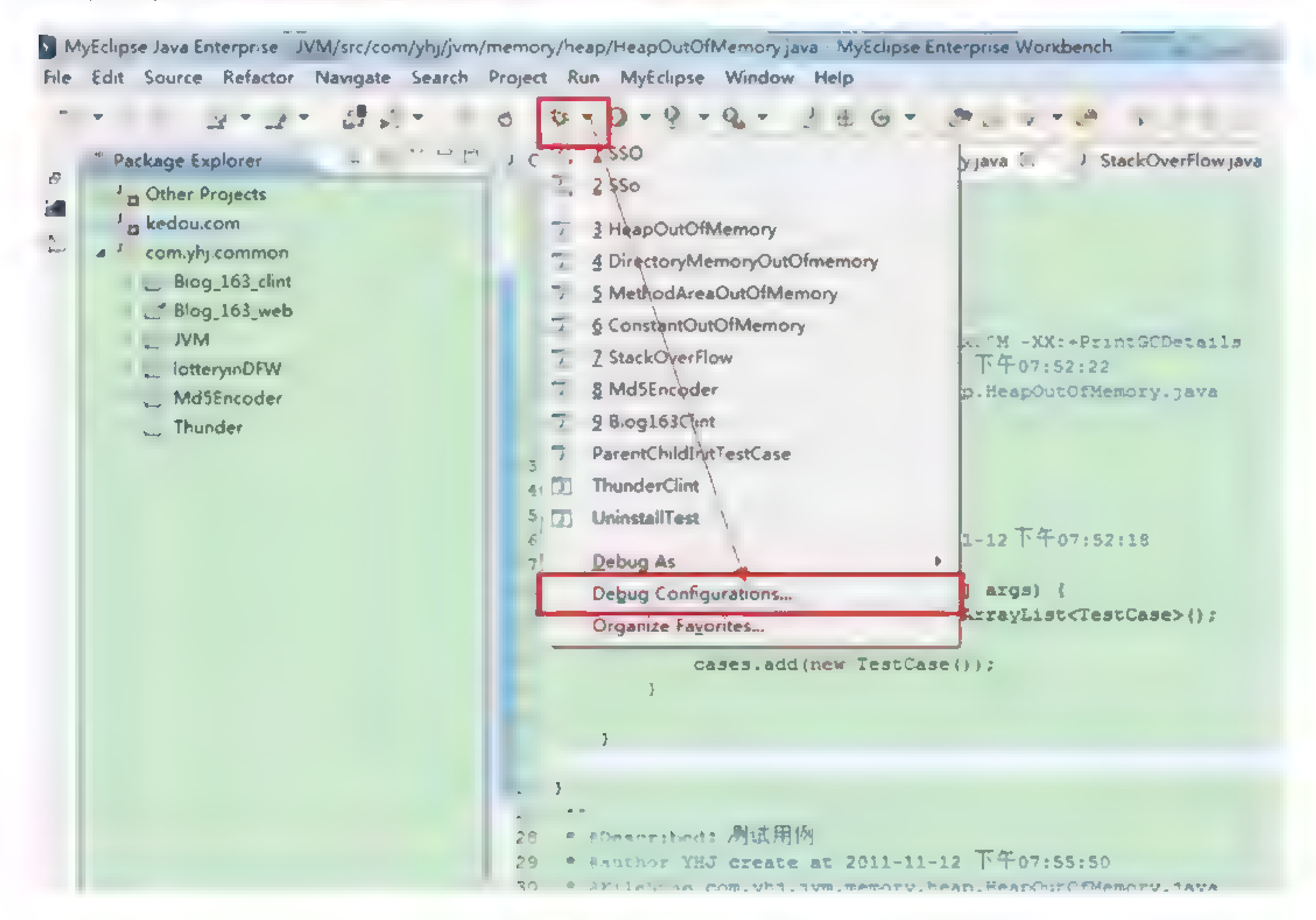


图 8-9 参数设置(1)

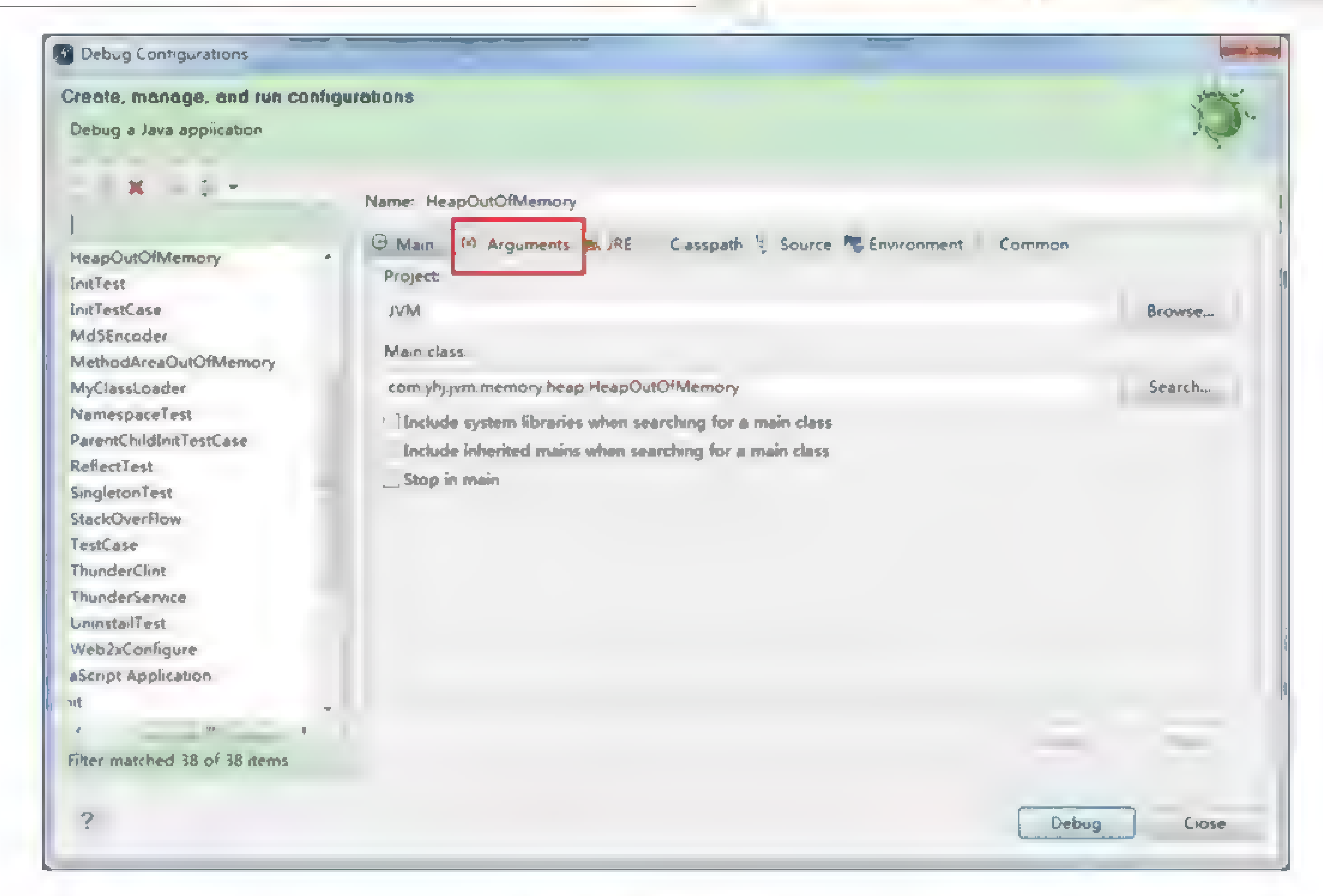


图 8-8　参数设置(2)

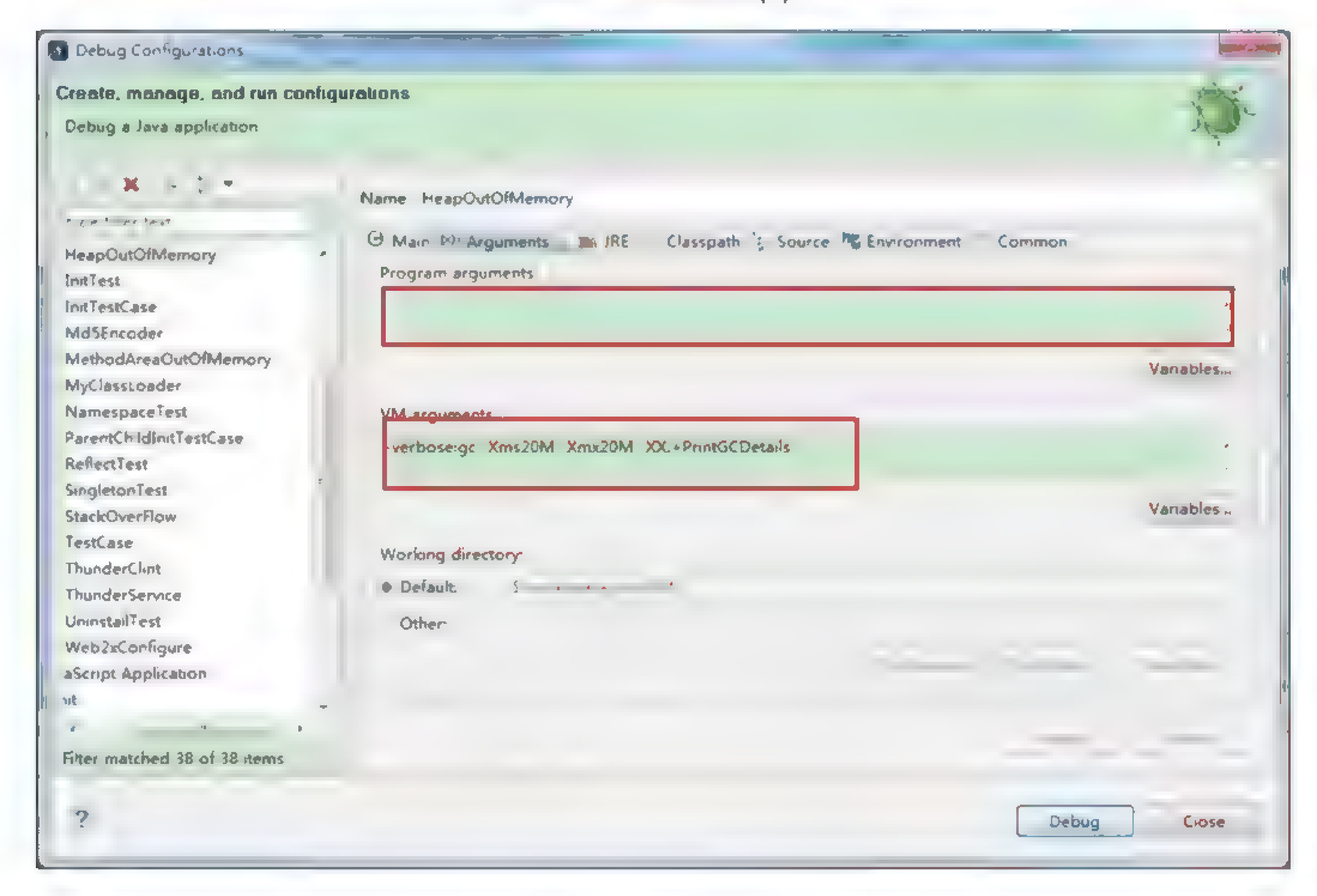


图 8-9　参数设置(3)

下面是测试代码：

```
package com.yhj.jvm.memory.heap;
import java.util.ArrayList;
import java.util.List;
public class HeapOutOfMemory {
    public static void main(String[] args) {
       List<TestCase> cases = new ArrayList<TestCase>();
       while(true){
          cases.add(new TestCase());
       }
    }
}
class TestCase{
}
```

运行后会输出：

```
    java.lang.OutOfMemoryError: Java heap space
    Dumping heap to java pid3404.hprof …
    Heap dump file created [22045981 bytes in 0.663 secs]
```

Java 堆内存的 OOM 异常是实际应用中最常见的内存溢出异常情况。出现 Java 堆内存溢出时，异常堆栈信息 java.lang.OutOfMemoryError 会跟着进一步提示“Java heap space”。

要解决这个区域的异常，一般的手段是首先通过内存映像分析工具(如 EclipseMemory Analyzer)对 dump 出来的堆转储快照进行分析，重点是确认内存中的对象是否是必要的，也就是要先分清楚到底是出现了内存泄露(Memory Leak)还是内存溢出(Memory Overflow)。如果是内存泄露，可进一步通过工具查看泄露对象到 GC Roots 的引用链。于是就能找到泄露对象是通过怎样的路径与 GC Roots 相关联并导致垃圾收集器无法自动回收它们的。掌握了泄露对象的类型信息，以及 GC Roots 引用链的信息，就可以比较准确地定位出泄露代码的位置。如果不存在泄露，换句话说就是内存中的对象确实都还必须存活着，那就应当检查虚拟机的堆参数(-Xmx 与-Xms)，与机器物理内存对比看是否还可以调大，从代码上检查是否存在某些对象生命周期过长、持有状态时间过长的情况，尝试减少程序运行期的内存消耗。

2. 虚拟机栈和本地方法栈溢出

由于在 HotSpot 虚拟机中并不区分虚拟机栈和本地方法栈，因此对于 HotSpot 来说，-Xoss 参数(设置本地方法栈大小)虽然存在，但实际上是无效的，栈容量只由-Xss 参数设定。关于虚拟机栈和本地方法栈，在 Java 虚拟机规范中描述了两种异常：

- 如果线程请求的栈深度大于虚拟机所允许的最大深度，将抛出 StackOverflowError 异常。
- 如果虚拟机在扩展栈时无法申请到足够的内存空间，则抛出 OutOfMemoryError 异常。

这里把异常分成两种情况看似更加严谨，但却存在着一些互相重叠的地方：当栈空间无法继续分配时，到底是内存太小，还是已使用的栈空间太大，其本质上只是对同一件事

情的两种描述而已。

在具体测试中会发现，如果将实验范围限制于单线程中的操作，尝试了下面两种方法均无法让虚拟机产生 OutOfMemoryError 异常，尝试的结果都是获得 StackOverflowError 异常。

例如下面的代码会造成栈溢出：

```
package com.yhj.jvm.memory.stack;
public class StackOverFlow {
    private int i ;
    public void plus() {
        i++;
        plus();
    }
    public static void main(String[] args) {
        StackOverFlow stackOverFlow = new StackOverFlow();
        try {
            stackOverFlow.plus();
        } catch (Exception e) {
            System.out.println("Exception:stack length:"+stackOverFlow.i);
            e.printStackTrace();
        } catch (Error e) {
            System.out.println("Error:stack length:"+stackOverFlow.i);
            e.printStackTrace();
        }
    }
}
```

通过测试表明：在单个线程下，无论是由于栈帧太大，还是虚拟机栈容量太小，当内存无法分配的时候，虚拟机抛出的都是 StackOverflowError 异常。如果测试时不限于单线程，通过不断地建立线程的方式倒是可以产生内存溢出异常。但是，这样产生的内存溢出异常与栈空间是否足够大并不存在任何联系，或者准确地说，在这种情况下，给每个线程的栈分配的内存越大，反而越容易产生内存溢出异常。原因其实不难理解，操作系统分配给每个进程的内存是有限制的，譬如 32 位的 Windows 限制为 2GB。虚拟机提供了参数来控制 Java 堆和方法区的这两部分内存的最大值。剩余的内存为 2GB(操作系统限制)减去 Xmx(最大堆容量)，再减去 MaxPermSize(最大方法区容量)，程序计数器消耗内存很小，可以忽略掉。如果虚拟机进程本身耗费的内存不计算在内，剩下的内存就由虚拟机栈和本地方法栈“瓜分”了。每个线程分配到的栈容量越大，可以建立的线程数量自然就越少，建立线程时就越容易把剩下的内存耗尽。这一点读者需要在开发多线程应用的时候特别注意，出现 StackOverflowError 异常时有错误堆栈可以阅读，相对来说，比较容易找到问题的所在。而且，如果使用虚拟机默认参数，栈深度在大多数情况下(因为每个方法压入栈的帧大小并不是一样的，所以只能说大多数情况下)达到 1000～2000 完全没有问题，对于正常的方法调用(包括递归)，这个深度应该完全够用了。但是，如果是建立过多线程导致的内存溢出，在不能减少线程数或者更换 64 位虚拟机的情况下，就只能通过减少最大堆和减少栈容量来换取更多的线程。如果没有这方面的经验，这种通过“减少内存”的手段来解决内存溢出的方式会比较难以想到。

例如下面的代码在创建线程时会导致 OOM 异常。

```
public class JavaVMStackOOM {
     private void dontStop() {
          while (true) {
          }
     }
     public void stackLeakByThread() {
          while (true) {
               Thread thread = new Thread(new Runnable() {
                    @Override
                    public void run() {
                         dontStop();
                    }
               });
               thread.start();
          }
     }
     public static void main(String[] args) throws Throwable {
          JavaVMStackOOM oom = new JavaVMStackOOM();
          oom.stackLeakByThread();
     }
}
```

如果读者要运行上面这段代码，记得要把当前工作存盘，上述代码执行时有很大令操作系统卡死的风险。运行后会输出：

```
Exception in thread "main" java.lang.OutOfMemoryError: unable to create new
native thread
```

3．运行时常量池

如果要向运行时常量池中添加内容，最简单的做法就是使用 String.intern()这个 Native 方法。该方法的作用是：如果池中已经包含一个等于此 String 对象的字符串，则返回代表池中这个字符串的 String 对象；否则，将此 String 对象包含的字符串添加到常量池中，并且返回此 String 对象的引用。由于常量池分配在方法区内，我们可以通过-XX:PermSize 和 -XX:MaxPermSize 限制方法区的大小，从而间接限制其中常量池的容量。例如下面的代码。

```
package com.yhj.jvm.memory.constant;
import java.util.ArrayList;
import java.util.List;
public class ConstantOutOfMemory {
   public static void main(String[] args) throws Exception {
      try {
         List<String> strings = new ArrayList<String>();
         int i = 0;
         while(true){
            strings.add(String.valueOf(i++).intern());
         }
      } catch (Exception e) {
         e.printStackTrace();
         throw e;
```

```
        }
    }
}
```

并且从运行结果中可以看到，运行时常量池溢出，在 OutOfMemoryError 后面跟随的提示信息是“PermGen space”，说明运行时常量池属于方法区(HotSpot 虚拟机中的永久代)的一部分。

4．方法区溢出

方法区用于存放 Class 相关信息，所以这个区域的测试我们借助 CGLib 直接操作字节码动态生成大量的 Class。值得注意的是，这里我们这个例子中模拟的场景其实经常会在实际应用中出现——当前很多主流框架，如 Spring、Hibernate 对类进行增强时，都会使用到 CGLib 这类字节码技术，当增强的类越多，就需要越大的方法区用于保证动态生成的 Class 可以加载入内存。

例如下面的代码会产生方法区溢出：

```
package com.yhj.jvm.memory.methodArea;
import java.lang.reflect.Method;
import net.sf.cglib.proxy.Enhancer;
import net.sf.cglib.proxy.MethodInterceptor;
import net.sf.cglib.proxy.MethodProxy;
public class MethodAreaOutOfMemory {
    public static void main(String[] args) {
        while(true){
            Enhancer enhancer = new Enhancer();
            enhancer.setSuperclass(TestCase.class);
            enhancer.setUseCache(false);
            enhancer.setCallback(new MethodInterceptor() {
                @Override
                public Object intercept(Object arg0, Method arg1, Object[] arg2,
                        MethodProxy arg3) throws Throwable {
                    return arg3.invokeSuper(arg0, arg2);
                }
            });
            enhancer.create();
        }
    }
}
class TestCase{
}
```

再看在下面的代码中，借助 CGLib 使得方法区出现 OOM 异常。

```
public class JavaMethodAreaOOM {
        public static void main(String[] args) {
                while (true) {
                        Enhancer enhancer = new Enhancer();
                        enhancer.setSuperclass(OOMObject.class);
                        enhancer.setUseCache(false);
                        enhancer.setCallback(new MethodInterceptor() {
                                public Object intercept(Object obj, Method method,
Object[] args, MethodProxy proxy) throws Throwable {
```

```
                    return proxy.invokeSuper(obj, args);
                }
            });
            enhancer.create();
        }
    }
    static class OOMObject {
    }
}
```

运行后会输出：

```
Caused by: java.lang.OutOfMemoryError: PermGen space
    at java.lang.ClassLoader.defineClass1(Native Method)
    at java.lang.ClassLoader.defineClassCond(ClassLoader.java:632)
    at java.lang.ClassLoader.defineClass(ClassLoader.java:616)
    ... 8 more
```

方法区溢出也是一种常见的内存溢出异常，一个类如果要被垃圾收集器回收掉，判定条件是非常苛刻的。在经常动态生成大量 Class 的应用中，需要特别注意类的回收状况。这类场景除了上面提到的程序使用了 CGLIB 字节码增强外，常见的还有：大量 JSP 或动态产生 JSP 文件的应用(JSP 第一次运行时需要编译为 Java 类)、基于 OSGi 的应用(即使是同一个类文件，被不同的加载器加载也会视为不同的类)等。

5. **本机直接内存**

DirectMemory 容量可通过-XX:MaxDirectMemorySize 指定，不指定的话默认与 Java 堆(-Xmx 指定)一样，下文代码越过了 DirectByteBuffer，直接通过反射获取 Unsafe 实例进行内存分配(Unsafe 类的 getUnsafe()方法限制了只有引导类加载器才会返回实例，也就是基本上只有 rt.jar 里面的类的才能使用)，因为 DirectByteBuffer 也会抛 OOM 异常，但抛出异常时实际上并没有真正向操作系统申请分配内存，而是通过计算得知无法分配既会抛出，真正申请分配的方法是 unsafe.allocateMemory()。

例如下面的代码：

```
public class DirectMemoryOOM {
    private static final int  1MB = 1024 * 1024;
    public static void main(String[] args) throws Exception {
        Field unsafeField = Unsafe.class.getDeclaredFields()[0];
        unsafeField.setAccessible(true);
        Unsafe unsafe = (Unsafe) unsafeField.get(null);
        while (true) {
            unsafe.allocateMemory( 1MB);
        }
    }
}
```

运行后会输出：

```
Exception in thread "main" java.lang.OutOfMemoryError
    at sun.misc.Unsafe.allocateMemory(Native Method)
    at org.fenixsoft.oom.DirectMemoryOOM.main(DirectMemoryOOM.java:20)
```

8.4 内 存 泄 露

在计算机科学中，内存泄露(Memory Leak)是指由于疏忽或错误造成程序未能释放已经不再使用的内存的情况。内存泄露并非指内存在物理上的消失，而是应用程序分配某段内存后，由于设计错误，失去了对该段内存的控制，因而造成了内存的浪费。内存泄露与许多其他问题有着相似的症状，并且通常情况下只能由那些可以获得程序源代码的程序员才可以分析出来。然而，有不少人习惯于把任何不需要的内存使用的增加描述为内存泄露，严格意义上来说这是不准确的。一般我们常说的内存泄露是指堆内存的泄露。堆内存是指程序从堆中分配的，大小任意的(内存块的大小可以在程序运行期决定)，使用完后必须显式释放的内存。应用程序一般使用 malloc、realloc、new 等函数从堆中分配到一块内存，使用完后，程序必须负责相应的调用 free 或 delete 释放该内存块，否则这块内存就不能被再次使用，我们就说这块内存泄露了。

8.4.1 内存泄露的分类

内存泄露通常分为以下四类。

1) 常发性内存泄露

发生内存泄露的代码会被多次执行到，每次被执行的时候都会导致一块内存泄露。

2) 偶发性内存泄露

发生内存泄露的代码只有在某些特定环境或操作过程下才会发生，常发性和偶发性是相对的。对于特定的环境，偶发性的也许就变成了常发性的。所以测试环境和测试方法对检测内存泄露至关重要。

3) 一次性内存泄露

发生内存泄露的代码只会被执行一次，或者由于算法上的缺陷，导致总会有一块且仅一块内存发生泄露。比如，在一个 Singleton 类的构造函数中分配内存，在析构函数中却没有释放该内存。而 Singleton 类只存在一个实例，所以内存泄露只会发生一次。

4) 隐式内存泄露

程序在运行过程中不停的分配内存，但是直到结束的时候才释放内存。严格地说这里并没有发生内存泄露，因为最终程序释放了所有申请的内存。但是对于一个服务器程序，需要运行几天，几周甚至几个月，不及时释放内存也可能导致最终耗尽系统的所有内存。所以，我们称这类内存泄露为隐式内存泄露。

8.4.2 内存泄露的定义

一般我们常说的内存泄露是指堆内存的泄露。堆内存是指程序从堆中分配的，大小任意的(内存块的大小可以在程序运行期决定)，使用完后必须显示释放的内存。应用程序一般使用 malloc、realloc、new 等函数从堆中分配到一块内存，使用完后，程序必须负责相应的调用 free 或 delete 释放该内存块，否则，这块内存就不能被再次使用，我们就说这块

内存泄露了。例如下面的这段小程序演示了堆内存发生泄露的情形：

```
void MyFunction(int nSize) {
char* p= new char[nSize];
if( !GetStringFrom( p, nSize ) ){
MessageBox( "Error" );
return; }
…
}
```

当函数 GetStringFrom()返回 0 的时候，指针 p 指向的内存就不会被释放。这是一种常见的发生内存泄露的情形。程序在入口处分配内存，在出口处释放内存，但是 c 函数可以在任何地方退出，所以一旦有某个出口处没有释放应该释放的内存，就会发生内存泄露。

8.4.3 内存泄露的常见问题和后果

内存泄露会因为减少可用内存的数量从而降低计算机的性能。最终在最糟糕的情况下，过多的可用内存被分配掉导致全部或部分设备停止正常工作，或者应用程序崩溃。内存泄露可能不严重，甚至能够被常规的手段检测出来。在现代操作系统中，一个应用程序使用的常规内存在程序终止时被释放。这表示一个短暂运行的应用程序中的内存泄露不会导致严重后果。在以下情况下，内存泄露会导致较严重的后果：

- 程序运行后置之不理，并且随着时间的流失，消耗越来越多的内存(比如服务器上的后台任务，尤其是嵌入式系统中的后台任务，这些任务可能被运行后很多年内都置之不理)；
- 新的内存被频繁地分配，比如当显示电脑游戏或动画视频画面时；
- 程序能够请求未被释放的内存(比如共享内存)，甚至是在程序终止的时候；
- 泄露在操作系统内部发生；
- 泄露在系统关键驱动中发生；
- 内存非常有限，比如在嵌入式系统或便携设备中；
- 当运行于一个终止时内存并不自动释放的操作系统(比如 AmigaOS)之上，而且一旦丢失只能通过重启来恢复。

内存泄露是程式设计中一项常见错误，特别是使用没有内置自动垃圾回收的编程语言，如 C 及 C++。一般情况下，内存泄露发生是因为不能存取动态分配的内存。目前有相当数量的调试工具用于检测不能存取的内存，从而可以防止内存泄露问题，如 IBM Rational Purify、BoundsChecker、Valgrind、Insure++及 memwatch 都是为 C/C++程式设计亦较受欢迎的内存除错工具。垃圾回收则可以应用到任何编程语言，而 C/C++也有此类函式库。

提供自动内存管理的编程语言如 Java、VB、.NET(.NET 内存泄露)以及 LISP，都不能避免内存泄露。例如，程式会把项目加入至列表，但在完成时没有移除，如同人把物件丢到一堆物品中或放到抽屉内，但后来忘记取走这件物品一样。内存管理器不能判断项目是否将再被存取，除非程式作出一些指示表明不会再被存取。

虽然内存管理器可以回复不能存取的内存，但它不可以释放可存取的内存，因为仍有

可能需要使用。现代的内存管理器因此为程式设计员提供技术来标示内存的可用性，以不同级别的“存取性”表示。内存管理器不会把需要存取可能较高的对象释放。当对象直接和一个强引用相关或者间接和一组强引用相关表示该对象存取性较强(强引用相对于弱引用，是防止对象被回收的一个引用)。要防止此类内存泄露，开发者必须使用对象后清理引用，一般都是在不再需要时将引用设成 null，如果有可能，把维持强引用的事件侦听器全部注销。

一般来说，自动内存管理对开发者来讲比较方便，因为他们不需要实现释放的动作，或担心清理内存的顺序，而不用考虑对象是否依然被引用。对开发者来说，了解一个引用是否有必要保持比了解一个对象是否被引用要简单得多。但是，自动内存管理不能消除所有的内容泄露。

如果一个程序存在内存泄露并且它的内存使用量稳定增长，通常不会有很快的症状。每个物理系统都有一个较大的内存量，如果内存泄露没有被中止(比如重启造成泄露的程序)的话，它迟早会造成问题。大多数的现代计算机操作系统都有存储在 RAM 芯片中主内存和存储在次级存储设备如硬盘中的虚拟内存，内存分配是动态的——每个进程根据要求获得相应的内存。存取活跃的页面文件被转移到主内存以提高存取速度；反之，存取不活跃的页面文件被转移到次级存储设备。当一个简单的进程消耗大量的内存时，它通常占用越来越多的主内存，使其他程序转到次级存储设备，使系统的运行效率大大降低。甚至在有内存泄露的程序终止后，其他程序需要相当长的时间才能切换到主内存，恢复原来的运行效率。

当系统所有的内存全部耗完后(包括主内存和虚拟内存，在嵌入式系统中，仅有主内存)，所有申请内存的操作将失败。这通常导致程序试图申请内存来终止自己，或造成分段内存访问错误(Segmentation Fault)。现在有一些专门为修复这种情况而设计的程序，常用的办法是预留一些内存。值得注意的是，第一个遭遇得不到内存问题的程序有时候并不是有内存泄露的程序。一些多任务操作系统有特殊的机制来处理内存耗尽的情况，如随机终止一个进程(可能会终止一些正常的进程)，或终止耗用内存最大的进程(很有可能是引起内存泄露的进程)。另一些操作系统则有内存分配限制，这样可以防止任何一个进程耗用完整个系统的内存。这种设计的缺点是有时候某些进程确实需要较大数量的内存时，如一些处理图像，视频和科学计算的进程，操作系统需要重新配置。如内存泄露发生在内核，表示操作系统自身发生了问题。那些没有完善的内存管理的计算机，如嵌入式系统，会因为一个长时间的内存泄露而崩溃。一些被公众访问的系统，如网络服务器或路由器很容易被黑客攻击，加入一段攻击代码，而产生内存泄露。

8.4.4　检测内存泄露

检测内存泄露的关键是要能截获住对分配内存和释放内存的函数的调用。截获住这两个函数，我们就能跟踪每一块内存的生命周期，比如每当成功的分配一块内存后，就把它的指针加入一个全局的 list 中；每当释放一块内存，再把它的指针从 list 中删除。这样当程序结束的时候，list 中剩余的指针就是指向那些没有被释放的内存。如果要检测堆内存的泄露，那么只需截获住 malloc/realloc/free 和 new/delete 即可。其实 new/delete 最终也是

用 malloc/free 的，所以只要截获前面一组即可。对于其他的泄露，可以采用类似的方法，截获住相应的分配和释放函数。比如要检测 BSTR 的泄露，就需要截获“SysAllocString/SysFreeString”；要检测 HMENU 的泄露，就需要截获“CreateMenu/DestroyMenu”。但是有的资源的分配函数有多个，释放函数只有一个，比如 SysAllocStringLen 也可以用来分配 BSTR，这时就需要截获多个分配函数。在 Windows 平台下，检测内存泄露的工具常用的一般有三种，MS C-Runtime Library 内建的检测功能；外挂式的检测工具有 Purify、BoundsChecker 等。这三种工具各有优缺点，MS C-Runtime Library 虽然功能上较之外挂式的工具要弱，但是它是免费的；Performance Monitor 虽然无法标示出发生问题的代码，但是它能检测出隐式的内存泄露的存在，这是其他两类工具无能为力的地方。

8.5 垃圾收集初探

本节将详细讲解 Java 虚拟机垃圾收集的基本知识，为读者学习本书后面的知识打下坚实的基础。

8.5.1 何谓垃圾收集

垃圾收集(Garbage Collection，本书简称 GC)提供了内存管理的机制，使得应用程序不需要在关注内存如何释放，内存用完后，垃圾收集会进行收集，这样就减轻了因为人为的管理内存而造成的错误，比如在 C++语言里，出现内存泄露时很常见的。

Java 语言是目前使用最多的依赖于垃圾收集器的语言，但是垃圾收集器策略从 20 世纪 60 年代就已经流行起来了，比如 Smalltalk 和 Eiffel 等编程语言也集成了垃圾收集器的机制。

在堆里面存放着 Java 世界中几乎所有的对象，在回收前首先要确定这些对象之中哪些还在存活，哪些已经“死”了，即不可能再被任何途径使用的对象。

8.5.2 常见的垃圾收集策略

所有的垃圾收集算法都面临同一个问题，那就是找出应用程序不可到达的内存块，将其释放，这里面得不可到达主要是指应用程序已经没有内存块的引用了，而在 Java 中，某个对象对应用程序是可到达的是指这个对象被根(根主要是指类的静态变量，或者活跃在所有线程栈的对象的引用)引用或者对象被另一个可到达的对象引用。

1. Reference Counting(引用计数)

引用计数是最简单直接的一种方式，这种方式在每一个对象中增加一个引用的计数，这个计数代表当前程序有多少个引用引用了此对象，如果此对象的引用计数变为 0，那么此对象就可以作为垃圾收集器的目标对象来收集。

这种策略的优点是简单、直接，不需要暂停整个应用；缺点是需要编译器的配合，编

译器要生成特殊的指令来进行引用计数的操作，比如每次将对象赋值给新的引用，或者者对象的引用超出了作用域等，并且不能处理循环引用的问题。

2. 跟踪收集器

跟踪收集器首先要暂停整个应用程序，然后开始从根对象扫描整个堆，判断扫描的对象是否有对象引用，在此需要搞清楚下面的三个问题。

(1) 如果每次扫描整个堆，那么势必让 GC 的时间变长，从而影响了应用本身的执行。因此在 JVM 里面采用了分代收集，在新生代收集的时候 minor gc 只需要扫描新生代，而不需要扫描老生代。

(2) JVM 采用了分代收集以后，minor gc 只扫描新生代，但是 minor gc 怎么判断是否有老生代的对象引用了新生代的对象，JVM 采用了卡片标记的策略，卡片标记将老生代分成了一块一块的，划分以后的每一个块就叫做一个卡片，JVM 采用卡表维护了每一个块的状态，当 Java 程序运行的时候，如果发现老生代对象引用或者释放了新生代对象的引用，那么就 JVM 就将卡表的状态设置为脏状态，这样每次 minor gc 的时候就会只扫描被标记为脏状态的卡片，而不需要扫描整个堆。

(3) GC 在收集一个对象的时候会判断是否有引用指向对象，在 Java 中的引用主要有 4 种，分别是 Strong reference、Soft reference、Weak reference 和 Phantom reference。

① Strong Reference

强引用是 Java 中默认采用的一种方式，我们平时创建的引用都属于强引用。如果一个对象没有强引用，那么对象就会被回收。例如：

```
public void testStrongReference(){
Object referent = new Object();
Object strongReference = referent;
referent = null;
System.gc();
assertNotNull(strongReference);
}
```

② Soft Reference

软引用的对象在 GC 的时候不会被回收，只有当内存不够用的时候才会真正的回收，因此软引用适合缓存的场合，这样使得缓存中的对象可以尽量的再内存中待长久一点。例如：

```
Public void testSoftReference(){
String  str =  "test";
SoftReference<String> softreference = new SoftReference<String>(str);
str=null;
System.gc();
assertNotNull(softreference.get());
}
```

③ Weak reference

弱引用有利于对象更快的被回收，假如一个对象没有强引用只有弱引用，那么在 GC 后，这个对象肯定会被回收。例如：

```
Public void testWeakReference(){
String  str = "test";
WeakReference<String> weakReference = new WeakReference<String>(str);
str=null;
System.gc();
assertNull(weakReference.get());
}
```

④ Phantom reference

- Mark-Sweep Collector(标记-清除收集器)

标记清除收集器最早由 Lisp 的发明人于 1960 年提出，标记清除收集器停止所有的工作，从根扫描每个活跃的对象，然后标记扫描过的对象，标记完成以后，清除那些没有被标记的对象。

优点是解决循环引用的问题，并且不需要编译器的配合，从而就不执行额外的指令。缺点是每个活跃的对象都要进行扫描，收集暂停的时间比较长。

- Copying Collector(复制收集器)

复制收集器将内存分为两块一样大小空间，某一个时刻，只有一个空间处于活跃的状态，当活跃的空间满的时候，GC 就会将活跃的对象复制到未使用的空间中去，原来不活跃的空间就变为了活跃的空间。

复制收集器的优点是只扫描可以到达的对象，不需要扫描所有的对象，从而减少了应用暂停的时间。缺点是需要额外的空间消耗，某一个时刻，总是有一块内存处于未使用状态。复制对象需要一定的开销。

- Mark-Compact Collector(标记-整理收集器)

标记整理收集器汲取了标记清除和复制收集器的优点，它分两个阶段执行：第一个阶段首先扫描所有活跃的对象，并标记所有活跃的对象；第二个阶段首先清除未标记的对象，然后将活跃的对象复制到堆得底部。Mark-compact 策略极大地减少了内存碎片，并且不需要像 Copy Collector 一样需要两倍的空间。

8.5.3　JVM 的垃圾收集策略

GC 执行时要耗费一定的 CPU 资源和时间，因此在 JDK1.2 以后，JVM 引入了分代收集的策略，其中对新生代采用“Mark-Compact“策略，而对老生代采用了“Mark-Sweep”的策略。其中新生代的垃圾收集器命名为“minor gc”，老生代的 GC 命名为“Full Gc”或“Major GC”。其中用 System.gc()强制执行的是 Full Gc。

1. Serial Collector

Serial Collector 是指任何时刻都只有一个线程进行垃圾收集，这种策略有一个名字“stop the whole world”，它需要停止整个应用的执行。这种类型的收集器适合于单 CPU 的机器。

```
Serial Copying Collector
```

此种 GC 用-XX:UseSerialGC 选项配置，它只用于新生代对象的收集。1.5.0 以后

-XX:MaxTenuringThreshold 来设置对象复制的次数。当 eden 空间不够的时候，GC 会将 eden 的活跃对象和一个名叫 From survivo 的空间中尚不够资格放入 Old 代的对象复制到另外一个名字叫 To Survivor 的空间。而此参数就是用来说明到底 From survivor 中的哪些对象不够资格，假如这个参数设置为 31，那么也就是说只有对象复制 31 次以后才算是有资格的对象。

From Survivor 和 To survivor 的角色是不断变化的，同一时间只有一块空间处于使用状态，这个空间就叫做 From Survivor 区，当复制一次后角色就发生了变化。

如果复制的过程中发现 To survivor 空间已经满了，那么就直接复制到 Old generation。

比较大的对象也会直接复制到 Old generation，在开发中，我们应该尽量避免这种情况的发生。

```
Serial  Mark-Compact Collector
```

串行的标记-整理收集器是 JDK5 update6 之前默认的老生代的垃圾收集器，此收集使得内存碎片最少化，但是它需要暂停的时间比较长。

2．Parallel Collector

Parallel Collector 主要是为了应对多 CPU、大数据量的环境。Parallel Collector 又可以分为以下两种。

(1) Parallel Copying Collector：此种 GC 用-XX:UseParNewGC 参数配置，它主要用于新生代的收集，此 GC 可以配合 CMS 一起使用。

(2) 在 1.4.1 版本以后用：

```
Parallel Mark-Compact Collector
```

此种 GC 用-XX:UseParallelOldGC 参数配置，此 GC 主要用于老生代对象的收集。1.6.0 后用：

```
Parallel scavenging Collector
```

此种 GC 用-XX:UseParallelGC 参数配置，它是对新生代对象的垃圾收集器，但是它不能和 CMS 配合使用，它适合于比较大新生代的情况，此收集器起始于 JDK 1.4.0。它比较适合于对吞吐量高于暂停时间的场合。

3．Concurrent Collector

Concurrent Collector 通过并行的方式进行垃圾收集，这样就减少了垃圾收集器收集一次的时间，这种 GC 在实时性要求高于吞吐量的时候比较有用。此种 GC 可以用参数-XX:UseConcMarkSweepGC 配置，此 GC 主要用于老生代和 Perm 代的收集。

8.6　对象的生死

在堆里面存放着 Java 世界中几乎所有的对象，在回收之前首先要确定这些对象之中哪些还在存活，哪些已经“死去”了，即不可能再被任何途径使用的对象。

8.6.1 引用计数算法(Reference Counting)

最初的想法，也是很多教科书判断对象是否存活的算法是这样的：给对象中添加一个引用计数器，当有一个地方引用它时计数器加 1，当引用失效时计数器减 1，任何时刻计数器为 0 的对象就是不可能再被使用的。

客观地说，引用计数算法实现简单，判定效率很高，在大部分情况下它都是一个不错的算法，但引用计数算法无法解决对象循环引用的问题。举个简单的例子：对象 A 和 B 分别有字段 b、a，令 A.b=B 和 B.a=A。除此之外，这两个对象再无任何引用，那实际上这两个对象已经不可能再被访问，但是引用计数算法却无法回收他们。

举一个简单的例子，请看代码清单 8-1 中的 testGC()方法：对象 objA 和 objB 都有字段 instance，赋值令 objA.instance = objB 及 objB.instance = objA，除此之外，这两个对象再无任何引用，实际上这两个对象已经不可能再被访问，但是它们因为互相引用着对方，导致它们的引用计数都不为 0，于是引用计数算法无法通知 GC 收集器回收它们。

代码清单 8-1

```
public class ReferenceCountingGC {
public Object instance = null;
private static final int  1MB = 1024 * 1024;
/**
* 这个成员属性的唯一意义就是占点内存，以便能在 GC 日志中看清楚是否被回收过
*/
private byte[] bigSize = new byte[2 *  1MB];
public static void testGC() {
    ReferenceCountingGC objA = new ReferenceCountingGC();
    ReferenceCountingGC objB = new ReferenceCountingGC();
    objA.instance = objB;
    objB.instance = objA;

    objA = null;
    objB = null;

    // 假设在这行发生 GC，那么 objA 和 objB 是否能被回收?
    System.gc();
}
}
```

运行结果：

```
[Full GC (System) [Tenured: 0K->210K(10240K), 0.0149142
secs] 4603K->210K(19456K), [Perm : 2999K->2999K(21248K)],
0.0150007 secs] [Times: user=0.01 sys=0.00, real=0.02 secs]
Heap
def new generation   total 9216K, used 82K
[0x00000000055e0000, 0x0000000005fe0000, 0x0000000005fe0000)
 Eden space 8192K,   1% used [0x00000000055e0000,
0x00000000055f4850, 0x0000000005de0000)
 from space 1024K,   0% used [0x0000000005de0000,
0x0000000005de0000, 0x0000000005ee0000)
 to   space 1024K,   0% used [0x0000000005ee0000,
```

```
0x000000005ee0000, 0x0000000005fe0000)
tenured generation   total 10240K, used 210K
[0x0000000005fe0000, 0x00000000069e0000, 0x00000000069e0000)
  the space 10240K,   2% used [0x0000000005fe0000,
0x0000000006014a18, 0x0000000006014c00, 0x00000000069e0000)
compacting perm gen  total 21248K, used 3016K
[0x00000000069e0000, 0x0000000007ea0000, 0x000000000bde0000)
  the space 21248K,  14% used [0x00000000069e0000,
0x0000000006cd2398, 0x0000000006cd2400, 0x0000000007ea0000)
No shared spaces configured.
```

从运行结果中可以清楚地看到 GC 日志中包含“4603K->210K”，意味着虚拟机并没有因为这两个对象互相引用就不回收它们，这也从侧面说明虚拟机并不是通过引用计数算法来判断对象是否存活的。

8.6.2　根搜索算法

在主流的商用程序语言中，例如 Java 和 C#都是使用根搜索算法(GC Roots Tracing)判定对象是否存活的。这个算法的基本思路就是通过一系列的名为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链(Reference Chain)，当一个对象到 GC Roots 没有任何引用链相连(用图论的话来说就是从 GC Roots 到这个对象不可达)时，则证明此对象是不可用的。如图 8-10 所示，对象 object 5、object 6、object 7 虽然互相有关联，但是它们到 GC Roots 是不可达的，所以它们将会被判定为是可回收的对象。

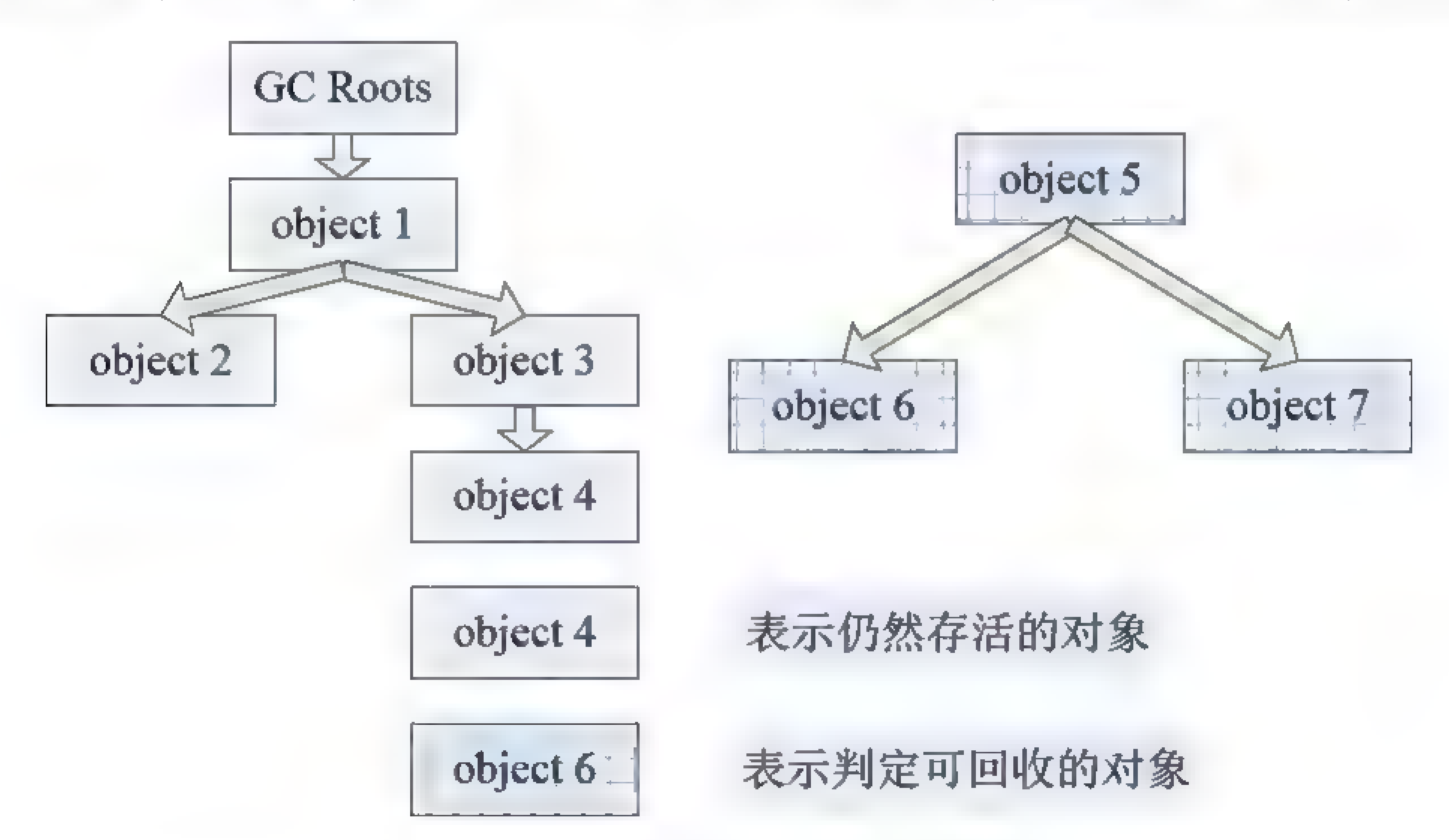


图 8-10　根搜索算法判定对象是否可回收

在 Java 语言里，可作为 GC Roots 的对象包括下面几种：

- 虚拟机栈(栈帧中的本地变量表)中的引用的对象。
- 方法区中的类静态属性引用的对象。
- 方法区中的常量引用的对象。
- 本地方法栈中 JNI(即一般说的 Native 方法)引用的对象。

8.6.3 再谈引用

无论是通过引用计数算法判断对象的引用数量，还是通过根搜索算法判断对象的引用链是否可达，判定对象是否存活都与“引用”有关。在 JDK 1.2 版本之前，Java 中的引用的定义很传统：如果 reference 类型的数据中存储的数值代表的是另外一块内存的起始地址，就称这块内存代表着一个引用。这种定义很纯粹，但是太过狭隘，一个对象在这种定义下只有被引用或者没有被引用两种状态，对于如何描述一些“食之无味，弃之可惜”的对象就显得无能为力。我们希望能描述这样一类对象：当内存空间还足够时，则能保留在内存之中；如果内存在进行垃圾收集后还是非常紧张，则可以抛弃这些对象。很多系统的缓存功能都符合这样的应用场景。

在 JDK 1.2 版本之后，Java 对引用的概念进行了扩充，将引用分为强引用(Strong Reference)、软引用(Soft Reference)、弱引用(Weak Reference)、虚引用(Phantom Reference)4种，这 4 种引用强度依次逐渐减弱。

强引用就是指在程序代码之中普遍存在的，类似“Object obj = new Object()”这类的引用，只要强引用还存在，垃圾收集器就永远不会回收掉被引用的对象。

软引用用来描述一些还有用，但并非必需的对象。对于软引用关联着的对象，在系统将要发生内存溢出异常之前，将会把这些对象列进回收范围之中并进行第二次回收。如果这次回收还是没有足够的内存，才会抛出内存溢出异常。在 JDK 1.2 之后，提供了 Soft Reference 类来实现软引用。

弱引用也是用来描述非必需对象的，但是它的强度比软引用更弱一些，被弱引用关联的对象只能生存到下一次垃圾收集发生之前。当垃圾收集器工作时，无论当前内存是否足够，都会回收掉只被弱引用关联的对象。在 JDK 1.2 之后，提供了 Weak Reference 类来实现弱引用。

虚引用也称为幽灵引用或者幻影引用，这是最弱的一种引用关系。一个对象是否有虚引用的存在，完全不会对其生存时间构成影响，也无法通过虚引用来取得一个对象实例。为一个对象设置虚引用关联的唯一目的就是希望能在这个对象被收集器回收时收到一个系统通知。在 JDK 1.2 之后，提供了 Phantom Reference 类来实现虚引用。

8.6.4 生存还是死亡

在根搜索算法中不可达的对象也并非是“非死不可”的，这时候它们暂时处于“缓刑”阶段，要真正宣告一个对象死亡，至少要经历两次标记过程：如果对象在进行根搜索后发现没有与 GC Roots 相连接的引用链，那它将会被第一次标记并且进行一次筛选，筛选的条件是此对象是否有必要执行 finalize()方法。当对象没有覆盖 finalize()方法，或者 finalize()方法已经被虚拟机调用过时，虚拟机将这两种情况都视为“没有必要执行”。

如果这个对象被判定为有必要执行 finalize()方法，那么这个对象将会被放置在一个名为 F-Queue 的队列之中，并在稍后由一条由虚拟机自动建立的、低优先级的 Finalizer 线程去执行。这里所谓的“执行”是指虚拟机会触发这个方法，但并不承诺会等待它运行结束。这样做的原因是，如果一个对象在 finalize()方法中执行缓慢，或者发生了死循环(更极

端的情况)，将很可能会导致 F-Queue 队列中的其他对象永久处于等待状态，甚至导致整个内存回收系统崩溃。finalize()方法是对象逃脱死亡命运的最后一次机会，稍后 GC 将对 F-Queue 中的对象进行第二次小规模的标记，如果对象要在 finalize()中成功拯救自己—只要重新与引用链上的任何一个对象建立关联即可，譬如把自己(this 关键字)赋值给某个类变量或对象的成员变量，那在第二次标记时它将被移除出“即将回收”的集合；如果对象这时候还没有逃脱，那它就真的离死不远了。从代码清单 8-2 中我们可以看到一个对象的 finalize()被执行，但是它仍然可以存活。

代码清单 8-2

```
/**
*此代码演示了两点：
*1.对象可以在被 GC 时自我拯救。
*2.这种自救的机会只有一次，因为一个对象的 finalize
()方法最多只会被系统自动调用一次
*@authorzzm
*/
publicclassFinalizeEscapeGC{
publicstaticFinalizeEscapeGCSAVE HOOK=null;
publicvoidisAlive(){
 System.out.println("yes,iamstillalive:)");
}
@Override
protectedvoidfinalize()throwsThrowable{
  super.finalize();
 System.out.println("finalizemehtodexecuted!");
 FinalizeEscapeGC.SAVE HOOK=this;
}
publicstaticvoidmain(String[]args)throwsThrowable{
 SAVE HOOK=newFinalizeEscapeGC();
 //对象第一次成功拯救自己
 SAVE HOOK=null;
 System.gc();
 //因为 Finalizer 方法优先级很低，暂停 0.5 秒，以等待它
 Thread.sleep(500);
 if(SAVE HOOK!=null){
 SAVE HOOK.isAlive();
 }else{
 System.out.println("no,iamdead:(");
 }

 //下面这段代码与上面的完全相同，但是这次自救却失败了
 SAVE HOOK=null;
 System.gc();
 //因为 Finalizer 方法优先级很低，暂停 0.5 秒，以等待它
 Thread.sleep(500);
 if(SAVE HOOK!=null){
 SAVE HOOK.isAlive();
 }else{
 System.out.println("no,iamdead:(");
 }
}
}
```

运行结果：

```
finalizemehtodexecuted!
yes,iamstillalive:)
no,iamdead:(
```

从代码清单 8-2 的运行结果可以看到，SAVE HOOK 对象的 finalize()方法确实被 GC 收集器触发过，并且在被收集前成功逃脱了。

另外一个值得注意的地方就是，代码中有两段完全一样的代码片段，执行结果却是一次逃脱成功，一次失败，这是因为任何一个对象的 finalize()方法都只会被系统自动调用一次，如果对象面临下一次回收，它的 finalize()方法不会被再次执行，因此第二段代码的自救行动失败了。

需要特别说明的是，上面关于对象“死亡”时 finalize()方法的描述可能带有悲情的艺术色彩，笔者并不鼓励大家使用这种方法来拯救对象。相反，笔者建议大家尽量避免使用它，因为它不是 C/C++中的析构函数，而是 Java 刚诞生时为了使 C/C++程序员更容易接受它所做出的一个妥协。它的运行代价高昂，不确定性大，无法保证各个对象的调用顺序。有些教材中提到它适合做“关闭外部资源”之类的工作，这完全是对这种方法的用途的一种自我安慰。finalize()能做的所有工作，使用 try-finally 或其他方式都可以做得更好、更及时，大家完全可以忘掉 Java 语言中还有这个方法的存在。

8.6.5　回收方法区

很多人认为方法区(或者 HotSpot 虚拟机中的永久代)是没有垃圾收集的，Java 虚拟机规范中确实说过可以不要求虚拟机在方法区实现垃圾收集，而且在方法区进行垃圾收集的“性价比”一般比较低：在堆中，尤其是在新生代中，常规应用进行一次垃圾收集一般可以回收 70%~95%的空间，而永久代的垃圾收集效率远低于此。

永久代的垃圾收集主要回收两部分内容：废弃常量和无用的类。回收废弃常量与回收 Java 堆中的对象非常类似。以常量池中字面量的回收为例，假如一个字符串“abc”已经进入了常量池中，但是当前系统没有任何一个 String 对象是叫做“abc”的，换句话说是没有任何 String 对象引用常量池中的“abc”常量，也没有其他地方引用了这个字面量，如果在这时候发生内存回收，而且必要的话，这个“abc”常量就会被系统“请”出常量池。常量池中的其他类(接口)、方法、字段的符号引用也与此类似。

判定一个常量是否是“废弃常量”比较简单，而要判定一个类是否是“无用的类”的条件则相对苛刻许多。类需要同时满足下面 3 个条件才能算是“无用的类”：

- 该类所有的实例都已经被回收，也就是 Java 堆中不存在该类的任何实例。
- 加载该类的 ClassLoader 已经被回收。
- 该类对应的 java.lang.Class 对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。

虚拟机可以对满足上述 3 个条件的无用类进行回收，这里说的仅仅是“可以”，而不是和对象一样，不使用了就必然会回收。是否对类进行回收，HotSpot 虚拟机提供了 -Xnoclassgc 参数进行控制，还可以使用 -verbose:class 及 -XX:+TraceClassLoading、


-XX:+TraceClassUnLoading 查看类的加载和卸载信息。

在大量使用反射、动态代理、CGLib 等 bytecode 框架的场景，以及动态生成 JSP 和 OSGi 这类频繁自定义 ClassLoader 的场景都需要虚拟机具备类卸载的功能，以保证永久代不会溢出。

8.7　垃圾收集算法

由于垃圾收集算法的实现涉及大量的程序细节，而且各个平台的虚拟机操作内存的方法又各不相同，因此本节不打算过多地讨论算法的实现，只是介绍几种算法的思想及其发展过程。

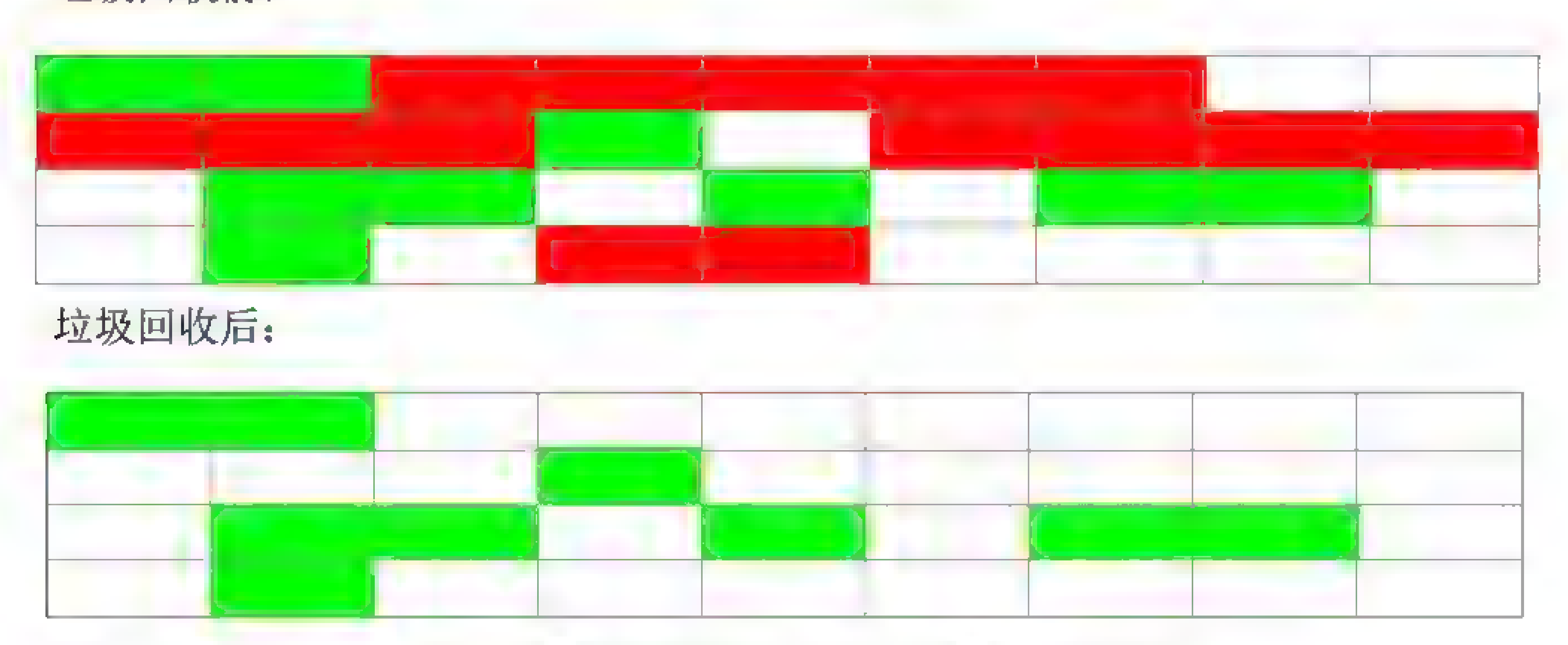
8.7.1　标记-清除算法

最基础的收集算法是“标记-清除”(Mark-Sweep)算法，如它的名字一样，算法分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，在标记完成后统一回收掉所有被标记的对象，它的标记过程其实在前一节讲述对象标记判定时已经基本介绍过了。之所以说它是最基础的收集算法，是因为后续的收集算法都是基于这种思路并对其缺点进行改进而得到的。它主要有两个缺点：一个是效率问题，标记和清除过程的效率都不高；另外一个是空间问题，标记清除之后会产生大量不连续的内存碎片，空间碎片太多可能会导致，当程序在以后的运行过程中需要分配较大对象时无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作。

标记-清除算法包括两个阶段：“标记”和“清除”。在标记阶段，确定所有要回收的对象，并做标记。清除阶段紧随标记阶段，将标记阶段确定不可用的对象清除。

标记-清除算法是基础的收集算法，标记和清除阶段的效率不高，而且清除后会产生大量的不连续空间，这样当程序需要分配大内存对象时，可能无法找到足够的连续空间。

垃圾回收前：



绿色：表示存活对象　红色：表示可回收对象　白色：表示未使用空间

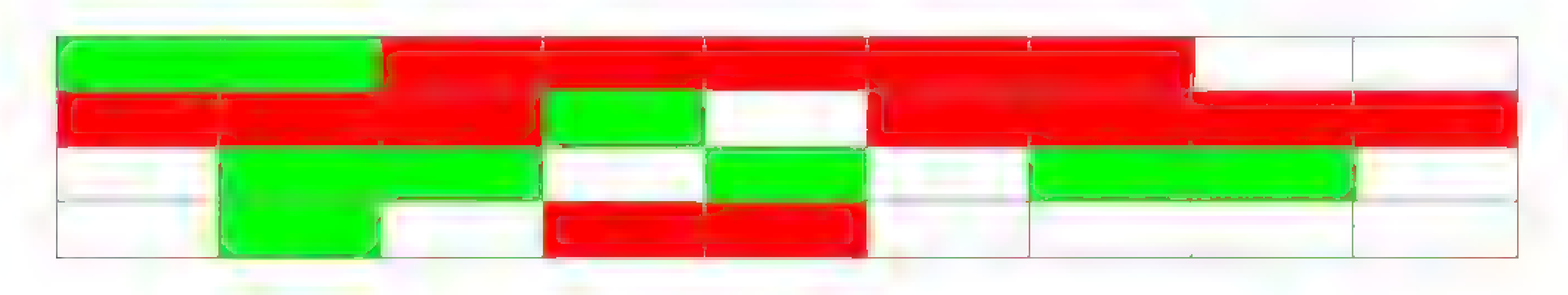
8.7.2 复制算法

为了解决效率问题，一种称为“复制”(Copying)的收集算法出现了，它将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。这样使得每次都是对其中的一块进行内存回收，内存分配时也就不用考虑内存碎片等复杂情况，只要移动堆顶指针，按顺序分配内存即可，实现简单，运行高效。只是这种算法的代价是将内存缩小为原来的一半，未免太高了一点。

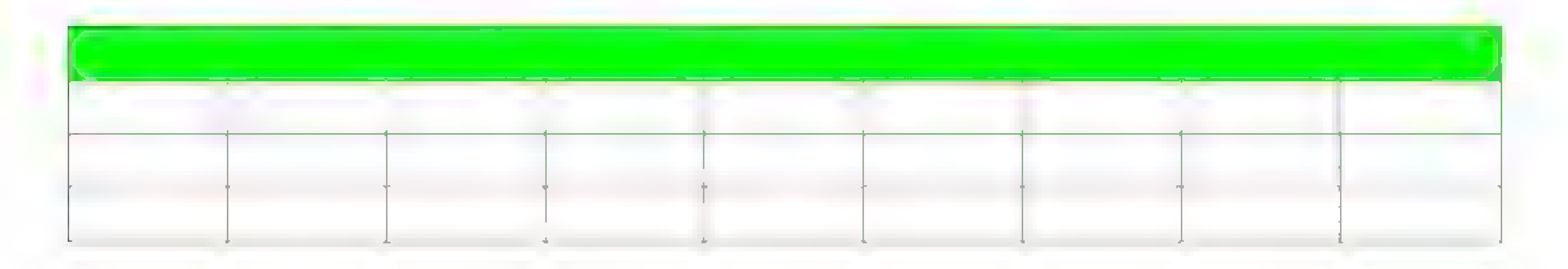
复制算法是把内存分成大小相等的两块，每次使用其中一块，当垃圾回收的时候，把存活的对象复制到另一块上，然后把这块内存整个清理掉。

复制算法实现简单，运行效率高，但是由于每次只能使用其中的一半，造成内存的利用率不高。现在的 JVM 用复制方法收集新生代，由于新生代中大部分对象(98%)都是朝生夕死的，所以两块内存的比例不是 1∶1，大概是 8∶1。

垃圾回收前：



垃圾回收后：



绿色：表示存活对象　红色：表示可回收对象　白色：表示未使用空间

现在的商业虚拟机都采用这种收集算法来回收新生代，IBM 的专门研究表明，新生代中的对象 98%是朝生夕死的，所以并不需要按照 1∶1 的比例来划分内存空间，而是将内存分为一块较大的 Eden 空间和两块较小的 Survivor 空间，每次使用 Eden 和其中的一块 Survivor。当回收时，将 Eden 和 Survivor 中还存活着的对象一次性地拷贝到另外一块 Survivor 空间上，最后清理掉 Eden 和刚才用过的 Survivor 的空间。HotSpot 虚拟机默认 Eden 和 Survivor 的大小比例是 8∶1，也就是每次新生代中可用内存空间为整个新生代容量的 90%(80%+10%)，只有 10%的内存是会被“浪费”的。当然，98%的对象可回收只是一般场景下的数据，我们没有办法保证每次回收都只有不多于 10%的对象存活，当 Survivor 空间不够用时，需要依赖其他内存(这里指老年代)进行分配担保(Handle Promotion)。

内存的分配担保就好比我们去银行借款，如果我们信誉很好，在 98%的情况下都能按时偿还，于是银行可能会默认我们下一次也能按时按量地偿还贷款，只需要有一个担保人能保证如果我不能还款时，可以从他的账户扣钱，那银行就认为没有风险了。内存的分配担保也一样，如果另外一块 Survivor 空间没有足够的空间存放上一次新生代收集下来的存活对象，这些对象将直接通过分配担保机制进入老年代。关于对新生代进行分配担保的内容，本章稍后在讲解垃圾收集器执行规则时还会再详细讲解。

8.7.3　标记-整理算法

复制收集算法在对象存活率较高时就要执行较多的复制操作，效率将会变低。更关键的是，如果不想浪费 50%的空间，就需要有额外的空间进行分配担保，以应对被使用的内存中所有对象都 100%存活的极端情况，所以在老年代一般不能直接选用这种算法。

根据老年代的特点，有人提出了另外一种标记-整理 (Mark-Compact)算法，标记过程仍然与标记-清除算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存。

标记-整理算法和标记-清除算法一样，但是标记-整理算法不是把存活对象复制到另一块内存，而是把存活对象往内存的一端移动，然后直接回收边界以外的内存。

标记-整理算法提高了内存的利用率，并且它适合在收集对象存活时间较长的老年代。

垃圾回收前：



垃圾回收后：



绿色：表示存活对象　　红色：表示可回收对象　　白色：表示未使用空间

8.7.4　分代收集算法

当前商业虚拟机的垃圾收集都采用“分代收集”(Generational Collection)算法，这种算法并没有什么新的思想，只是根据对象的存活周期的不同将内存划分为几块。一般是把 Java 堆分为新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法。在新生代中，每次垃圾收集时都发现有大批对象死去，只有少量存活，那就选用复制算法，

只需要付出少量存活对象的复制成本就可以完成收集。而老年代中因为对象存活率高、没有额外空间对它进行分配担保，就必须使用标记 清理或标记 整理算法来进行回收。

分代收集是根据对象的存活时间把内存分为新生代和老年代，根据个代对象的存活特点，每个代采用不同的垃圾回收算法。新生代采用标记 复制算法，老年代采用标记 整理算法。

8.8 垃圾收集器

如果说收集算法是内存回收的方法论，垃圾收集器就是内存回收的具体实现。Java 虚拟机规范中对垃圾收集器应该如何实现并没有任何规定，因此不同的厂商、不同版本的虚拟机所提供的垃圾收集器都可能会有很大的差别，并且一般都会提供参数供用户根据自己的应用特点和要求组合出各个年代所使用的收集器。这里讨论的收集器基于 Sun HotSpot 虚拟机 1.6 版 Update 22，这个虚拟机包含的所有收集器如图 8-11 所示。

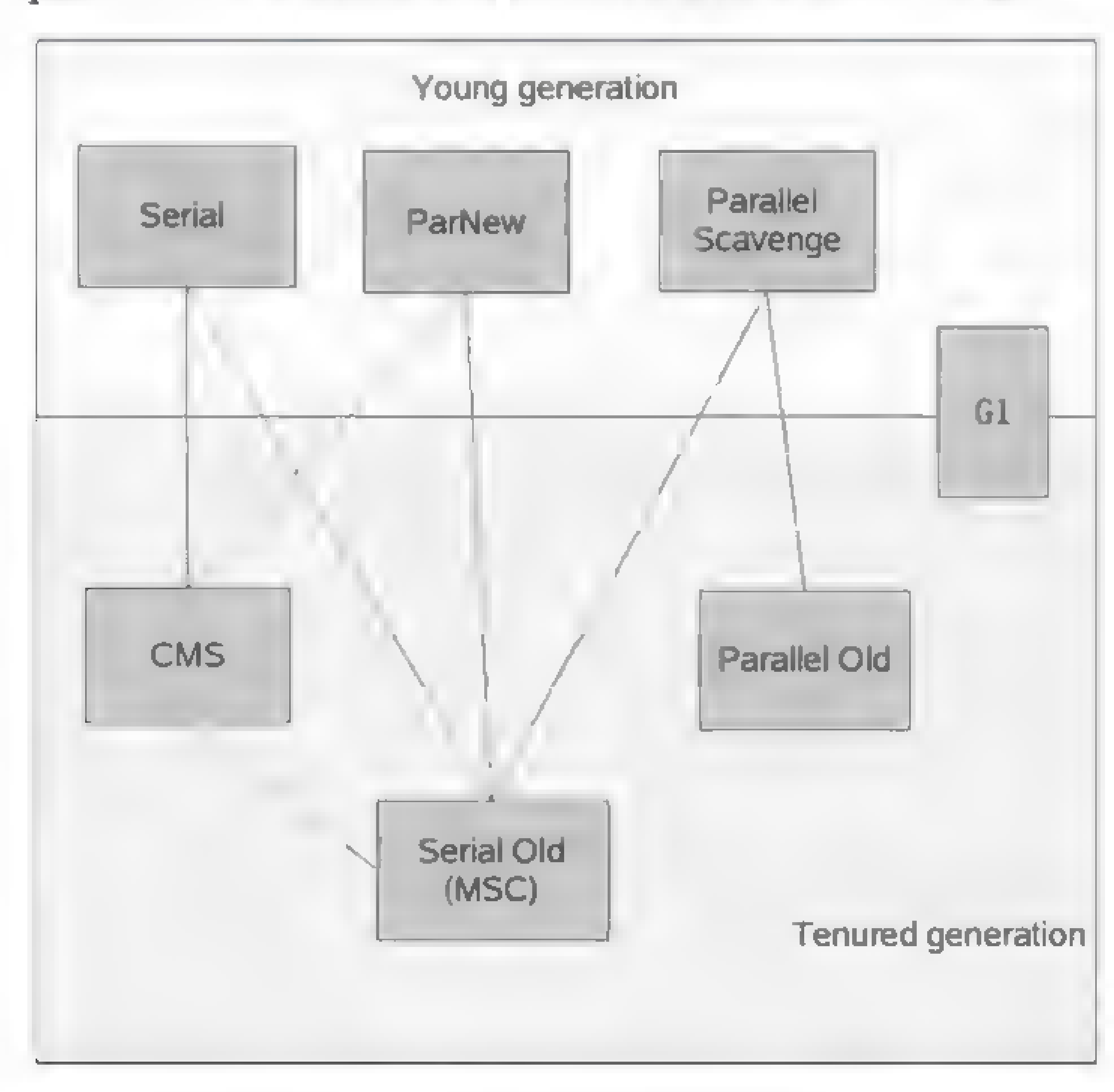


图 8-11 HotSpot JVM1.6 的垃圾收集器

图 8-11 展示了 7 种作用于不同分代的收集器(包括 JDK 1.6_Update14 后引入的 Early Access 版 G1 收集器)，如果两个收集器之间存在连线，就说明它们可以搭配使用。

在介绍这些收集器各自的特性之前，我们先来明确一个观点：虽然我们是在对各个收集器进行比较，但并非为了挑选一个最好的收集器出来。因为直到现在为止还没有最好的收集器出现，更加没有万能的收集器，所以我们选择的只是对具体应用最合适的收集器。

这点不需要多加解释就能证明：如果有一种放之四海皆准、任何场景下都适用的完美收集器存在，那 HotSpot 虚拟机就没必要实现那么多不同的收集器了。

8.8.1 Serial 收集器

Serial 收集器是最基本、历史最悠久的收集器，曾经(在 JDK 1.3.1 之前)是虚拟机新生代收集的唯一选择。大家看名字就知道，这个收集器是一个单线程的收集器，但它的“单线程”的意义并不仅仅是说明它只会使用一个 CPU 或一条收集线程去完成垃圾收集工作，更重要的是在它进行垃圾收集时，必须暂停其他所有的工作线程(Sun 将这件事情称之为“Stop The World”)，直到它收集结束。“Stop The World”这个名字也许听起来很酷，但这项工作实际上是由虚拟机在后台自动发起和自动完成的，在用户不可见的情况下把用户的正常工作的线程全部停掉，这对很多应用来说都是难以接受的。你想想，要是你的电脑每运行一个小时就会暂停响应 5 分钟，你会有什么样的心情。

对于“Stop The World”带给用户的恶劣体验，虚拟机的设计者们表示完全理解，但也表示非常委屈：“你妈妈在给你打扫房间的时候，肯定也会让你老老实实地在椅子上或房间外待着，如果她一边打扫，你一边乱扔纸屑，这房间还能打扫完吗？”这确实是一个合情合理的矛盾，虽然垃圾收集这项工作听起来和打扫房间属于一个性质的，但实际上肯定还要比打扫房间复杂得多啊！

从 JDK 1.3 开始，一直到现在还没正式发布的 JDK 1.7，HotSpot 虚拟机开发团队为消除或减少工作线程因内存回收而导致停顿的努力一直在进行着，从 Serial 收集器到 Parallel 收集器，再到 Concurrent Mark Sweep(CMS)现在还未正式发布的 Garbage First(G1)收集器，我们看到了一个个越来越优秀(也越来越复杂)的收集器的出现，用户线程的停顿时间在不断缩短，但是仍然没有办法完全消除(这里暂不包括 RTSJ 中的收集器)。寻找更优秀的垃圾收集器的工作仍在继续！

写到这里，笔者似乎已经把 Serial 收集器描述成一个老而无用，食之无味弃之可惜的鸡肋了，但实际上到现在为止，它依然是虚拟机运行在 Client 模式下的默认新生代收集器。它也有着优于其他收集器的地方：简单而高效(与其他收集器的单线程比)。对于限定单个 CPU 的环境来说，Serial 收集器由于没有线程交互的开销，专心做垃圾收集自然可以获得最高的单线程收集效率。在用户的桌面应用场景中，分配给虚拟机管理的内存一般来说不会很大，收集几十兆甚至一两百兆的新生代(仅仅是新生代使用的内存，桌面应用基本上不会再大了)，停顿时间完全可以控制在几十毫秒最多一百多毫秒以内，只要不是频繁发生，这点停顿是可以接受的。所以，Serial 收集器对于运行在 Client 模式下的虚拟机来说是一个很好的选择。

8.8.2 ParNew 收集器

ParNew 收集器其实就是 Serial 收集器的多线程版本，除了使用多条线程进行垃圾收集之外，其余行为包括 Serial 收集器可用的所有控制参数(例如：-XX:SurvivorRatio、-XX:PretenureSizeThreshold、-XX:HandlePromotionFailure 等)、收集算法、Stop The World、对象分配规则、回收策略等都与 Serial 收集器完全一样，实现上这两种收集器也共

用了相当多的代码。

ParNew 收集器除了多线程收集之外，其他与 Serial 收集器相比并没有太多创新之处，但它却是许多运行在 Server 模式下的虚拟机中首选的新生代收集器。其中有一个与性能无关但很重要的原因是，除了 Serial 收集器外，目前只有它能与 CMS 收集器配合工作。在 JDK 1.5 时期，HotSpot 推出了一款在强交互应用中几乎可称为有划时代意义的垃圾收集器——CMS 收集器，这款收集器是 HotSpot 虚拟机中第一款真正意义上的并发(Concurrent)收集器，它第一次实现了让垃圾收集线程与用户线程(基本上)同时工作，用前面那个例子的话来说，就是做到了在你妈妈打扫房间的时候你还能同时往地上扔纸屑。

不幸的是，它作为老年代的收集器，却无法与 JDK 1.4.0 中已经存在的新生代收集器 Parallel Scavenge 配合工作，所以在 JDK 1.5 中使用 CMS 来收集老年代的时候，新生代只能选择 ParNew 或 Serial 收集器中的一个。ParNew 收集器也是使用 -XX:+UseConcMarkSweepGC 选项后的默认新生代收集器，也可以使用 -XX:+UseParNewGC 选项来强制指定它。

ParNew 收集器在单 CPU 的环境中绝对不会有比 Serial 收集器更好的效果，甚至由于存在线程交互的开销，该收集器在通过超线程技术实现的两个 CPU 的环境中都不能百分之百地保证能超越 Serial 收集器。当然，随着可以使用的 CPU 的数量的增加，它对于 GC 时系统资源的利用还是很有好处的。它默认开启的收集线程数与 CPU 的数量相同，在 CPU 非常多(譬如 32 个，现在 CPU 动辄就 4 核加超线程，服务器超过 32 个逻辑 CPU 的情况越来越多了)的环境下，可以使用-XX:ParallelGCThreads 参数来限制垃圾收集的线程数。

注意：从 ParNew 收集器开始，后面还将会接触到几款并发和并行的收集器。在大家可能产生疑惑之前，有必要先解释两个名词：并发和并行。这两个名词都是并发编程中的概念，在谈论垃圾收集器的上下文语境中，他们可以解释为：

- 并行(Parallel)：指多条垃圾收集线程并行工作，但此时用户线程仍然处于等待状态。
- 并发(Concurrent)：指用户线程与垃圾收集线程同时执行(但不一定是并行的，可能会交替执行)，用户程序继续运行，而垃圾收集程序运行于另一个 CPU 上。

8.8.3 Parallel Scavenge 收集器

Parallel Scavenge 收集器也是一个新生代收集器，它也是使用复制算法的收集器，又是并行的多线程收集器……看上去和 ParNew 都一样，那它有什么特别之处呢？

Parallel Scavenge 收集器的特点是它的关注点与其他收集器不同，CMS 等收集器的关注点尽可能地缩短垃圾收集时用户线程的停顿时间，而 Parallel Scavenge 收集器的目标则是达到一个可控制的吞吐量(Throughput)。所谓吞吐量就是 CPU 用于运行用户代码的时间与 CPU 总消耗时间的比值，即吞吐量 = 运行用户代码时间 /(运行用户代码时间 + 垃圾收集时间)，虚拟机总共运行了 100 分钟，其中垃圾收集花掉 1 分钟，那吞吐量就是 99%。

停顿时间越短就越适合需要与用户交互的程序，良好的响应速度能提升用户的体验；而高吞吐量则可以最高效率地利用 CPU 时间，尽快地完成程序的运算任务，主要适合在后台运算而不需要太多交互的任务。

Parallel Scavenge 收集器提供了两个参数用于精确控制吞吐量，分别是控制最大垃圾收集停顿时间的-XX:MaxGCPauseMillis 参数及直接设置吞吐量大小的 -XX:GCTimeRatio 参数。

MaxGCPauseMillis 参数允许的值是一个大于 0 的毫秒数，收集器将尽力保证内存回收花费的时间不超过设定值。不过大家不要异想天开地认为如果把这个参数的值设置得稍小一点就能使得系统的垃圾收集速度变得更快，GC 停顿时间缩短是以牺牲吞吐量和新生代空间来换取的：系统把新生代调小一些，收集 300MB 新生代肯定比收集 500MB 快吧，这也直接导致垃圾收集发生得更频繁一些，原来 10 秒收集一次、每次停顿 100 毫秒，现在变成 5 秒收集一次、每次停顿 70 毫秒。停顿时间的确在下降，但吞吐量也降下来了。

GCTimeRatio 参数的值应当是一个大于 0 小于 100 的整数，也就是垃圾收集时间占总时间的比率，相当于是吞吐量的倒数。如果把此参数设置为 19，那允许的最大 GC 时间就占总时间的 5%(即 1 /(1+19))，默认值为 99，就是允许最大 1%(即 1/(1+99))的垃圾收集时间。

由于与吞吐量关系密切，Parallel Scavenge 收集器也经常被称为“吞吐量优先”收集器。除上述两个参数之外，Parallel Scavenge 收集器还有一个参数-XX:+UseAdaptiveSizePolicy 值得关注。这是一个开关参数，当这个参数打开之后，就不需要手工指定新生代的大小(-Xmn)、Eden 与 Survivor 区的比例(-XX:SurvivorRatio)、晋升老年代对象年龄(-XX:PretenureSizeThreshold)等细节参数了，虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最合适的停顿时间或最大的吞吐量，这种调节方式称为 GC 自适应的调节策略(GC Ergonomics)。如果读者对于收集器运作原理不太了解，手工优化存在困难的时候，使用 Parallel Scavenge 收集器配合自适应调节策略，把内存管理的调优任务交给虚拟机去完成将是一个很不错的选择。只需要把基本的内存数据设置好(如-Xmx 设置最大堆)，然后使用 MaxGCPauseMillis 参数(更关注最大停顿时间)或 GCTimeRatio 参数(更关注吞吐量)给虚拟机设立一个优化目标，那具体细节参数的调节工作就由虚拟机完成了。自适应调节策略也是 Parallel Scavenge 收集器与 ParNew 收集器的一个重要区别。

8.8.4　Serial Old 收集器

Serial Old 是 Serial 收集器的老年代版本，它同样是一个单线程收集器，使用标记-整理算法。这个收集器的主要意义也是被 Client 模式下的虚拟机使用。如果在 Server 模式下，它主要还有两大用途：一个是在 JDK 1.5 及之前的版本中与 Parallel Scavenge 收集器搭配使用，另外一个就是作为 CMS 收集器的后备预案，在并发收集发生 Concurrent Mode Failure 的时候使用。

8.8.5　Parallel Old 收集器

Parallel Old 是 Parallel Scavenge 收集器的老年代版本，使用多线程和标记-整理算法。这个收集器是在 JDK 1.6 中才开始提供的，在此之前，新生代的 Parallel Scavenge 收集器一直处于比较尴尬的状态。原因是，如果新生代选择了 Parallel Scavenge 收集器，老年代

除了 Serial Old(PS MarkSweep)收集器外别无选择(还记得上面说过 Parallel Scavenge 收集器无法与 CMS 收集器配合工作吗)。由于单线程的老年代 Serial Old 收集器在服务端应用性能上的“拖累”，即便使用了 Parallel Scavenge 收集器也未必能在整体应用上获得吞吐量最大化的效果，又因为老年代收集中无法充分利用服务器多 CPU 的处理能力，在老年代很大而且硬件比较高级的环境中，这种组合的吞吐量甚至还不一定有 ParNew 加 CMS 的组合“给力”。

直到 Parallel Old 收集器出现后，“吞吐量优先”收集器终于有了比较名副其实的应用组合，在注重吞吐量及 CPU 资源敏感的场合，都可以优先考虑 Parallel Scavenge 加 Parallel Old 收集器。

8.8.6 CMS 收集器

CMS(Concurrent Mark Sweep)收集器是一种以获取最短回收停顿时间为目标的收集器。目前很大一部分的 Java 应用都集中在互联网站或 B/S 系统的服务端上，这类应用尤其重视服务的响应速度，希望系统停顿时间最短，以给用户带来较好的体验。CMS 收集器就非常符合这类应用的需求。

从名字(包含“Mark Sweep”)上就可以看出 CMS 收集器是基于标记-清除算法实现的，它的运作过程相对于前面几种收集器来说要更复杂一些，整个过程分为 4 个步骤，包括：

- 初始标记(CMS initial mark)。
- 并发标记(CMS concurrent mark) 。
- 重新标记(CMS remark)。
- 并发清除(CMS concurrent sweep)。

其中初始标记、重新标记这两个步骤仍然需要“Stop The World”。初始标记仅仅只是标记一下 GC Roots 能直接关联到的对象，速度很快，并发标记阶段就是进行 GC Roots Tracing 的过程，而重新标记阶段则是为了修正并发标记期间，因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段稍长一些，但远比并发标记的时间短。

由于整个过程中耗时最长的并发标记和并发清除过程中，收集器线程都可以与用户线程一起工作，所以总体上来说，CMS 收集器的内存回收过程是与用户线程一起并发地执行的。

CMS 是一款优秀的收集器，它的最主要优点在名字上已经体现出来了：并发收集、低停顿，Sun 的一些官方文档里面也称之为并发低停顿收集器(Concurrent Low Pause Collector)。但是 CMS 还远达不到完美的程度，它有以下三个显著的缺点。

(1) CMS 收集器对 CPU 资源非常敏感。其实，面向并发设计的程序都对 CPU 资源比较敏感。在并发阶段，它虽然不会导致用户线程停顿，但是会因为占用了一部分线程(或者说 CPU 资源)而导致应用程序变慢，总吞吐量会降低。CMS 默认启动的回收线程数是(CPU 数量+3)/4，也就是当 CPU 在 4 个以上时，并发回收时垃圾收集线程最多占用不超过 25% 的 CPU 资源。但是当 CPU 不足 4 个时(譬如 2 个)，那么 CMS 对用户程序的影响就可能变

得很大，如果 CPU 负载本来就比较大的时候，还分出一半的运算能力去执行收集器线程，就可能导致用户程序的执行速度忽然降低了 50%，这也很让人受不了。为了解决这种情况，虚拟机提供了一种称为"增量式并发收集器"(Incremental Concurrent Mark Sweep/i-CMS)的 CMS 收集器变种，所做的事情和单 CPU 年代 PC 机操作系统使用抢占式来模拟多任务机制的思想一样，就是在并发标记和并发清理的时候让 GC 线程、用户线程交替运行，尽量减少 GC 线程的独占资源的时间，这样整个垃圾收集的过程会更长，但对用户程序的影响就会显得少一些，速度下降也就没有那么明显，但是目前版本中，i-CMS 已经被声明为"deprecated"，即不再提倡用户使用。

(2) CMS 收集器无法处理浮动垃圾(Floating Garbage)，可能出现"Concurrent Mode Failure"失败而导致另一次 Full GC 的产生。由于 CMS 并发清理阶段用户线程还在运行着，伴随程序的运行自然还会有新的垃圾不断产生，这一部分垃圾出现在标记过程之后，CMS 无法在本次收集中处理掉它们，只好留待下一次 GC 时再将其清理掉。这一部分垃圾就称为"浮动垃圾"。也是由于在垃圾收集阶段用户线程还需要运行，即还需要预留足够的内存空间给用户线程使用，因此 CMS 收集器不能像其他收集器那样等到老年代几乎完全被填满了再进行收集，需要预留一部分空间提供并发收集时的程序运作使用。在默认设置下，CMS 收集器在老年代使用了 68%的空间后就会被激活，这是一个偏保守的设置，如果在应用中老年代增长不是太快，可以适当调高参数-XX:CMSInitiatingOccupancyFraction 的值来提高触发百分比，以便降低内存回收次数以获取更好的性能。要是 CMS 运行期间预留的内存无法满足程序需要，就会出现一次"Concurrent Mode Failure"失败，这时候虚拟机将启动后备预案：临时启用 Serial Old 收集器来重新进行老年代的垃圾收集，这样停顿时间就很长了。所以说参数-XX:CMSInitiatingOccupancyFraction 设置得太高将会很容易导致大量 Concurrent Mode Failure 失败，性能反而降低。

(3) 还有最后一个缺点，在本节在开头说过，CMS 是一款基于标记-清除算法实现的收集器，如果读者对前面这种算法介绍还有印象的话，就可能想到这意味着收集结束时会产生大量空间碎片。空间碎片过多时，将会给大对象分配带来很大的麻烦，往往会出现老年代还有很大的空间剩余，但是无法找到足够大的连续空间来分配当前对象，不得不提前触发一次 Full GC。为了解决这个问题，CMS 收集器提供了一个-XX:+UseCMSCompactAtFullCollection 开关参数，用于在"享受"完 Full GC 服务之后额外免费附送一个碎片整理过程，内存整理的过程是无法并发的。空间碎片问题没有了，但停顿时间不得不变长了。虚拟机设计者还提供了另外一个参数-XX:CMSFullGCsBeforeCompaction，这个参数用于设置在执行多少次不压缩的 Full GC 后，跟着来一次带压缩的。

8.8.7 G1 收集器

G1(Garbage First)收集器是当前收集器技术发展的最前沿成果，在 JDK 1.6_Update14 中提供了 Early Access 版本的 G1 收集器以供试用。在将来 JDK 1.7 正式发布的时候，G1 收集器很可能会有一个成熟的商用版本随之发布。这里只对 G1 收集器进行简单介绍。

G1 收集器是垃圾收集器理论进一步发展的产物，它与前面的 CMS 收集器相比有两个

显著的改进：一是G1收集器是基于标记-整理算法实现的收集器，也就是说它不会产生空间碎片，这对于长时间运行的应用系统来说非常重要。二是它可以非常精确地控制停顿，既能让使用者明确指定在一个长度为M毫秒的时间片段内，消耗在垃圾收集上的时间不得超过N毫秒，这几乎已经是实时Java(RTSJ)的垃圾收集器的特征了。

G1收集器可以实现在基本不牺牲吞吐量的前提下完成低停顿的内存回收，这是由于它能够极力地避免全区域的垃圾收集，之前的收集器进行收集的范围都是整个新生代或老年代，而G1将整个Java堆(包括新生代、老年代)划分为多个大小固定的独立区域(Region)，并且跟踪这些区域里面的垃圾堆积程度，在后台维护一个优先列表，每次根据允许的收集时间，优先回收垃圾最多的区域(这就是Garbage First名称的来由)。区域划分及有优先级的区域回收，保证了G1收集器在有限的时间内可以获得最高的收集效率。

8.8.8 垃圾收集器参数总结

JDK 1.6中的各种垃圾收集器到此已全部介绍完毕，在描述过程中提到了很多虚拟机非稳定的运行参数，表8-1整理了这些参数以供读者实践时参考。

表8-1 垃圾收集相关的常用参数

参 数	描 述
UseSerialGC	虚拟机运行在Client模式下的默认值，打开此开关后，使用Serial +Serial Old的收集器组合进行内存回收
UseParNewGC	打开此开关后，使用ParNew + Serial Old的收集器组合进行内存回收
UseConcMarkSweepGC	打开此开关后，使用ParNew + CMS + Serial Old的收集器组合进行内存回收。Serial Old收集器将作为CMS收集器出现Concurrent Mode Failure失败后的后备收集器使用
UseParallelGC	虚拟机运行在Server模式下的默认值，打开此开关后，使用Parallel Scavenge + Serial Old(PS MarkSweep)的收集器组合进行内存回收
UseParallelOldGC	打开此开关后，使用Parallel Scavenge + Parallel Old的收集器组合进行内存回收
SurvivorRatio	新生代中Eden区域与Survivor区域的容量比值，默认为8，代表Eden ：Survivor=8：1
PretenureSizeThreshold	直接晋升到老年代的对象大小，设置这个参数后，大于这个参数的对象将直接在老年代分配
MaxTenuringThreshold	晋升到老年代的对象年龄。每个对象在坚持过一次Minor GC之后，年龄就加1，当超过这个参数值时就进入老年代
UseAdaptiveSizePolicy	动态调整Java堆中各个区域的大小以及进入老年代的年龄
HandlePromotionFailure	是否允许分配担保失败，即老年代的剩余空间不足以应付新生代的整个Eden和Survivor区的所有对象都存活的极端情况
ParallelGCThreads	设置并行GC时进行内存回收的线程数

续表

参　数	描　述
GCTimeRatio	GC 时间占总时间的比率，默认值为 99，即允许 1% 的 GC 时间。仅在使用 Parallel Scavenge 收集器时生效
MaxGCPauseMillis	设置 GC 的最大停顿时间。仅在使用 Parallel Scavenge 收集器时生效
CMSInitiatingOccupancyFraction	设置 CMS 收集器在老年代空间被使用多少后触发垃圾收集。默认值为 68%，仅在使用 CMS 收集器时生效
UseCMSCompactAtFullCollection	设置 CMS 收集器在完成垃圾收集后是否要进行一次内存碎片整理。仅在使用 CMS 收集器时生效
CMSFullGCsBeforeCompaction	设置 CMS 收集器在进行若干次垃圾收集后再启动一次内存碎片整理。仅在使用 CMS 收集器时生效

8.9　内存分配与回收策略

Java 技术体系中所提倡的自动内存管理最终可以归结为自动化地解决了两个问题：给对象分配内存和回收分配给对象的内存。关于回收内存这一点，我们已经使用了大量的篇幅去介绍虚拟机中的垃圾收集器体系及其运作原理，现在我们再一起来探讨一下给对象分配内存的那点事儿。

对象的内存分配，往大方向上讲，就是在堆上分配(但也可能经过 JIT 编译后被拆散为标量类型并间接地在栈上分配)，对象主要分配在新生代的 Eden 区上，如果启动了本地线程分配缓冲，将按线程优先在 TLAB 上分配。少数情况下也可能会直接分配在老年代中，分配的规则并不是百分之百固定的，其细节取决于当前使用的是哪一种垃圾收集器组合，还有虚拟机中与内存相关的参数的设置。

接下来我们将会讲解几条最普遍的内存分配规则，并通过代码去验证这些规则。本节中的代码在测试时使用 Client 模式虚拟机运行，没有手工指定收集器组合，换句话说，验证的是使用 Serial/Serial Old 收集器下(ParNew/Serial Old 收集器组合的规则也基本一致)的内存分配和回收的策略。读者不妨根据自己项目中使用的收集器写一些程序去验证一下使用其他几种收集器的内存分配策略。

8.9.1　对象优先在 Eden 分配

大多数情况下，对象在新生代 Eden 区中分配。当 Eden 区没有足够的空间进行分配时，虚拟机将发起一次 Minor GC。

虚拟机提供了-XX:+PrintGCDetails 这个收集器日志参数，告诉虚拟机在发生垃圾收集行为时打印内存回收日志，并且在进程退出的时候输出当前内存各区域的分配情况。在实际应用中，内存回收日志一般是打印到文件后通过日志工具进行分析，不过本实验的日志并不多，直接阅读就能看得很清楚。

代码清单 8-3 的 testAllocation()方法中，尝试分配 3 个 2MB 大小和 1 个 4MB 大小的对象，在运行时通过-Xms20M、-Xmx20M 和 -Xmn10M 这 3 个参数限制 Java 堆大小为 20MB，且不可扩展，其中 10MB 分配给新生代，剩下的 10MB 分配给老年代。-XX:SurvivorRatio 8 决定了新生代中 Eden 区与一个 Survivor 区的空间比例是 8 比 1，从输出的结果也能清晰地看到“eden space 8192K、from space 1024K、to space 1024K”的信息，新生代总可用空间为 9216KB(Eden 区+1 个 Survivor 区的总容量)。

执行 testAllocation()中分配 allocation4 对象的语句时会发生一次 Minor GC，这次 GC 的结果是新生代 6651KB 变为 148KB，而总内存占用量则几乎没有减少(因为 allocation1、2、3 三个对象都是存活的，虚拟机几乎没有找到可回收的对象)。这次 GC 发生的原因是给 allocation4 分配内存的时候，发现 Eden 已经被占用了 6MB，剩余空间已不足以分配 allocation4 所需的 4MB 内存，因此发生 Minor GC。GC 期间虚拟机又发现已有的 3 个 2MB 大小的对象全部无法放入 Survivor 空间(Survivor 空间只有 1MB 大小)，所以只好通过分配担保机制提前转移到老年代去。

这次 GC 结束后，4MB 的 allocation4 对象被顺利分配在 Eden 中。因此程序执行完的结果是 Eden 占用 4MB(被 allocation4 占用)、Survivor 空闲、老年代被占用 6MB(被 allocation1、2、3 占用)。通过 GC 日志可以证实这一点。

新生代 GC(Minor GC)：指发生在新生代的垃圾收集动作，因为 Java 对象大多都具备朝生夕灭的特性，所以 Minor GC 非常频繁，一般回收速度也比较快。

老年代 GC(Major GC/Full GC)：指发生在老年代的 GC，出现了 Major GC，经常会伴随至少一次的 Minor GC。当然这并非百分百绝对的，在 ParallelScavenge 收集器的收集策略里就有直接进行 Major GC 的策略选择过程。MajorGC 的速度一般会比 Minor GC 慢 10 倍以上。

代码清单 8-3

```
private static final int  1MB = 1024 * 1024;

/**
* VM 参数：-verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:SurvivorRatio=8
*/
public static void testAllocation() {
   byte[] allocation1, allocation2, allocation3, allocation4;
   allocation1 = new byte[2 *  1MB];
   allocation2 = new byte[2 *  1MB];
   allocation3 = new byte[2 *  1MB];
   allocation4 = new byte[4 *  1MB];  // 出现一次 Minor GC
}
```

运行结果：

```
[GC [DefNew: 6651K->148K(9216K), 0.0070106 secs]
6651K->6292K(19456K), 0.0070426 secs] [Times:
user=0.00 sys=0.00, real=0.00 secs]
Heap
def new generation   total 9216K, used 4326K
[0x029d0000, 0x033d0000, 0x033d0000)
 eden space 8192K,  51% used [0x029d0000, 0x02de4828, 0x031d0000)
```

```
 from space 1024K,  14% used [0x032d0000, 0x032f5370, 0x033d0000)
 to   space 1024K,   0% used [0x031d0000, 0x031d0000, 0x032d0000)
tenured generation  total 10240K, used 6144K [0x033d0000, 0x03dd0000, 0x03dd0000)
  the space 10240K,  60% used [0x033d0000, 0x039d0030, 0x039d0200, 0x03dd0000)
compacting perm gen  total 12288K, used 2114K [0x03dd0000, 0x049d0000, 0x07dd0000)
  the space 12288K,  17% used [0x03dd0000, 0x03fe0998, 0x03fe0a00, 0x049d0000)
No shared spaces configured.
```

8.9.2　大对象直接进入老年代

所谓大对象就是指，需要大量连续内存空间的 Java 对象，最典型的大对象就是那种很长的字符串及数组。大对象对虚拟机的内存分配来说就是一个坏消息，经常出现大对象容易导致内存还有不少空间时就提前触发垃圾收集以获取足够的连续空间来“安置”它们。

虚拟机提供了一个-XX:PretenureSizeThreshold 参数，令大于这个设置值的对象直接在老年代中分配。这样做的目的是避免在 Eden 区及两个 Survivor 区之间发生大量的内存拷贝。

执行代码清单 8-4 中的 testPretenureSizeThreshold()方法后，我们看到 Eden 空间几乎没有被使用，而老年代 10MB 的空间被使用了 40%，也就是 4MB 的 allocation 对象直接就分配在老年代中，这是因为 PretenureSizeThreshold 被设置为 3MB(就是 3145728B，这个参数不能与-Xmx 之类的参数一样直接写 3MB)，因此超过 3MB 的对象都会直接在老年代中进行分配。

注意：PretenureSizeThreshold 参数只对 Serial 和 ParNew 两款收集器有效，Parallel Scavenge 收集器不认识这个参数，Parallel Scavenge 收集器一般并不需要设置。如果遇到必须使用此参数的场合，可以考虑 ParNew 加 CMS 的收集器组合。

代码清单 8-4

```
private static final int  1MB = 1024 * 1024;

/**
* VM 参数：-verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:SurvivorRatio=8
* -XX:PretenureSizeThreshold=3145728
*/
public static void testPretenureSizeThreshold() {
byte[] allocation;
allocation = new byte[4 *  1MB];  //直接分配在老年代中
}
```

运行结果：

```
Heap
def new generation   total 9216K, used 671K
[0x029d0000, 0x033d0000, 0x033d0000)
 eden space 8192K,   8% used [0x029d0000, 0x02a77e98, 0x031d0000)
 from space 1024K,   0% used [0x031d0000, 0x031d0000, 0x032d0000)
 to   space 1024K,   0% used [0x032d0000, 0x032d0000, 0x033d0000)
tenured generation   total 10240K, used 4096K [0x033d0000, 0x03dd0000,
0x03dd0000)
  the space 10240K,  40% used [0x033d0000, 0x037d0010, 0x037d0200,
```

```
0x03dd0000)
compacting perm gen total 12288K, used 2107K [0x03dd0000, 0x049d0000, 0x07dd0000)
  the space 12288K,  17% used [0x03dd0000, 0x03fdefd0, 0x03fdf000, 0x049d0000)
No shared spaces configured.
```

8.9.3 长期存活的对象将进入老年代

虚拟机既然采用了分代收集的思想来管理内存，那内存回收时就必须能识别哪些对象应当放在新生代，哪些对象应放在老年代中。为了做到这点，虚拟机给每 个对象定义了一个对象年龄(Age)计数器。如果对象在 Eden 出生并经过第一次 Minor GC 后仍然存活，并且能被 Survivor 容纳的话，将被移动到 Survivor 空间中，并将对象年龄设为 1。对象在 Survivor 区中每熬过一次 Minor GC，年龄就增加 1 岁，当它的年龄增加到一定程度(默认为 15 岁)时，就会被晋升到老年代中。对象晋升老年代的年龄阈值，可以通过参数-XX:MaxTenuringThreshold 来设置

读者可以试试分别以-XX:MaxTenuringThreshold=1 和-XX:MaxTenuringThreshold=15 两种设置来执行代码清单 8-5 中的 testTenuringThreshold()方法，此方法中 allocation1 对象需要 256KB 的内存空 间，Survivor 空间可以容纳。当 MaxTenuringThreshold=1 时，allocation1 对象在第二次 GC 发生时进入老年代，新生 代已使用的内存 GC 后会非常干净地变成 0KB。而 MaxTenuringThreshold=15 时，第二次 GC 发生后，allocation1 对象则还留在新生代的 Survivor 空间，这时候新生代仍然有 404KB 的空间被占用。

代码清单 8-5

```
private static final int  1MB = 1024 * 1024;
/**
* VM 参数：-verbose:gc -Xms20M -Xmx20M -Xmn10M
-XX:SurvivorRatio=8 -XX:MaxTenuringThreshold=1
* -XX:+PrintTenuringDistribution
*/
@SuppressWarnings("unused")
public static void testTenuringThreshold() {
 byte[] allocation1, allocation2, allocation3;
 allocation1 = new byte[ 1MB / 4];
  // 什么时候进入老年代取决于 XX:MaxTenuringThreshold 设置
 allocation2 = new byte[4 *  1MB];
 allocation3 = new byte[4 *  1MB];
 allocation3 = null;
 allocation3 = new byte[4 *  1MB];
}
```

以 MaxTenuringThreshold=1 的参数设置来运行的结果：

```
[GC [DefNew
Desired Survivor size 524288 bytes, new threshold 1 (max 1)
- age   1:     414664 bytes,     414664 total : 4859K->404K(9216K),
0.0065012 secs] 4859K->4500K
(19456K), 0.0065283 secs] [Times: user=0.02 sys-0.00, real=0.02 secs]
[GC [DefNew Desired Survivor size 524288 bytes, new threshold 1 (max 1) :
4500K >0K(9216K), 0.0009253 secs] 8596K >4500K
(19456K), 0.0009458 secs] [Times: user 0.00 sys 0.00, real 0.00 secs]
```

```
Heap
def new generation   total 9216K, used 4178K
[0x029d0000, 0x033d0000, 0x033d0000)
 eden space 8192K,  51% used [0x029d0000, 0x02de4828, 0x031d0000)
 from space 1024K,   0% used [0x031d0000,0x031d0000, 0x032d0000)
 to   space 1024K,   0% used [0x032d0000, 0x032d0000, 0x033d0000)
tenured generation  total 10240K, used 4500K [0x033d0000, 0x03dd0000, 0x03dd0000)
  the space 10240K,  43% used [0x033d0000, 0x03835348, 0x03835400, 0x03dd0000)
compacting perm gen total 12288K, used 2114K [0x03dd0000, 0x049d0000, 0x07dd0000)
  the space 12288K,  17% used [0x03dd0000, 0x03fe0998, 0x03fe0a00, 0x049d0000)
No shared spaces configured.
```

以 MaxTenuringThreshold=15 的参数设置来运行的结果：

```
[GC [DefNew
Desired Survivor size 524288 bytes, new threshold 15 (max 15)
- age   1:     414664 bytes,     414664 total  : 4859K->404K(9216K),
0.0049637 secs] 4859K->
4500K(19456K), 0.0049932 secs] [Times: user=0.00 sys=0.00, real=0.00
secs]
[GC [DefNew Desired Survivor size 524288 bytes, new threshold 15 (max 15)
- age   2:     414520 bytes,     414520 total : 4500K->404K(9216K),
0.0008091 secs] 8596K->
4500K(19456K), 0.0008305 secs] [Times: user=0.00 sys=0.00, real=0.00
secs]
Heap
def new generation   total 9216K, used 4582K
[0x029d0000, 0x033d0000, 0x033d0000)
 eden space 8192K,  51% used [0x029d0000, 0x02de4828, 0x031d0000)
 from space 1024K,  39% used [0x031d0000, 0x03235338, 0x032d0000)
 to   space 1024K,   0% used [0x032d0000, 0x032d0000, 0x033d0000)
tenured generation   total 10240K, used 4096K [0x033d0000, 0x03dd0000,
0x03dd0000)
  the space 10240K,  40% used [0x033d0000, 0x037d0010, 0x037d0200,
0x03dd0000)
compacting perm gen  total 12288K, used 2114K [0x03dd0000, 0x049d0000,
0x07dd0000)
  the space 12288K,  17% used [0x03dd0000,0x03fe0998, 0x03fe0a00,
0x049d0000)
No shared spaces configured.
```

8.9.4 动态对象年龄判定

为了能更好地适应不同程序的内存状况，虚拟机并不总是要求对象的年龄必须达到 MaxTenuringThreshold 才能晋升老年代，如果在 Survivor 空间中相同年龄所有对象大小的总和大于 Survivor 空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年代，无须等到 MaxTenuringThreshold 中要求的年龄。

执行代码清单 8-6 中的 testTenuringThreshold2() 方法，并设置参数 -XX:MaxTenuringThreshold=15，会发现运行结果中 Survivor 的空间占用仍然为 0%，而老年代比预期增加了 6%，也就是说 allocation1、allocation2 对象都直接进入了老年代，而没有等到 15 岁的临界年龄。因为这两个对象加起来已经达到了 512KB，并且它们是同年的，满

足同年对象达到 Survivor 空间的一半规则。我们只要注释掉其中一个对象的 new 操作，就会发现另外一个不会晋升到老年代中去了。

代码清单 8-6

```
private static final int  1MB = 1024 * 1024;
/**
* VM 参数：-verbose:gc -Xms20M -Xmx20M -Xmn10M
-XX:SurvivorRatio=8 -XX:MaxTenuringThreshold=15
* -XX:+PrintTenuringDistribution
*/
@SuppressWarnings("unused")
public static void testTenuringThreshold2() {
 byte[] allocation1, allocation2, allocation3, allocation4;
 allocation1 = new byte[ 1MB / 4];
  // allocation1+allocation2 大于 survivor 空间的一半
 allocation2 = new byte[ 1MB / 4];
 allocation3 = new byte[4 *  1MB];
 allocation4 = new byte[4 *  1MB];
 allocation4 = null;
 allocation4 = new byte[4 *  1MB];
}
```

运行结果：

```
[GC [DefNew
Desired Survivor size 524288 bytes, new threshold 1 (max 15)
- age   1:    676824 bytes,    676824 total : 5115K->660K(9216K), 0.0050136
secs] 5115K->
4756K(19456K), 0.0050443 secs] [Times: user=0.00
sys=0.01, real=0.01 secs]
[GC [DefNew
Desired Survivor size 524288 bytes, new threshold 15 (max 15)
: 4756K->0K(9216K), 0.0010571 secs] 8852K->4756K
(19456K), 0.0011009 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
Heap
def new generation   total 9216K, used 4178K [0x029d0000, 0x033d0000,
0x033d0000)
 eden space 8192K,  51% used [0x029d0000, 0x02de4828, 0x031d0000)
 from space 1024K,   0% used [0x031d0000, 0x031d0000, 0x032d0000)
 to   space 1024K,   0% used [0x032d0000, 0x032d0000, 0x033d0000)
tenured generation   total 10240K, used 4756K [0x033d0000, 0x03dd0000,
0x03dd0000)
  the space 10240K,  46% used [0x033d0000, 0x038753e8, 0x03875400, 0x03dd0000)
compacting perm gen  total 12288K, used 2114K [0x03dd0000, 0x049d0000,
0x07dd0000)
  the space 12288K,  17% used [0x03dd0000, 0x03fe09a0, 0x03fe0a00, 0x049d0000)
No shared spaces configured.
```

8.9.5 空间分配担保

在发生 Minor GC 时，虚拟机会检测之前每次晋升到老年代的平均大小是否大于老年代的剩余空间大小，如果大于，则改为直接进行一次 Full GC；如果小于，则查看 HandlePromotionFailure 设置是否允许担保失败；如果允许，那只会进行 Minor GC；如果

不允许，则也要改为进行一次 Full GC。

前面提到过，新生代使用复制收集算法，但为了内存利用率，只使用其中一个 Survivor 空间来作为轮换备份，因此当出现大量对象在 Minor GC 后仍然存活的情况时，最极端就是内存回收后新生代中所有对象都存活，就需要老年代进行分配担保，让 Survivor 无法容纳的对象直接进入老年代。与生活中的贷款担保类似，老年代要进行这样的担保，前提是老年代本身还有容纳这些对象的剩余空间，一共有多少对象会活下来，在实际完成内存回收之前是无法明确知道的，所以只好取之前每一次回收晋升到老年代对象容量的平均大小值作为经验值，与老年代的剩余空间进行比较，决定是否进行 Full GC 来让老年代腾出更多空间。

取平均值进行比较其实仍然是一种动态概率的手段，也就是说如果某次 Minor GC 存活后的对象突增，远远高于平均值的话，依然会导致担保失败(Handle Promotion Failure)。如果出现了 HandlePromotionFailure 失败，那就只好在失败后重新发起一次 Full GC。虽然担保失败时绕的圈子是最大的，但大部分情况下都还是会将 HandlePromotionFailure 开关打开，避免 Full GC 过于频繁，参见代码清单 8-7。

代码清单 8-7

```
private static final int  1MB = 1024 * 1024;
/**
* VM 参数：-verbose:gc -Xms20M -Xmx20M -Xmn10M
-XX:SurvivorRatio=8 -XX:-
HandlePromotionFailure
*/
@SuppressWarnings("unused")
public static void testHandlePromotion() {
 byte[] allocation1, allocation2, allocation3,
allocation4, allocation5, allocation6, allocation7;
 allocation1 = new byte[2 *  1MB];
 allocation2 = new byte[2 *  1MB];
 allocation3 = new byte[2 *  1MB];
 allocation1 = null;
 allocation4 = new byte[2 *  1MB];
 allocation5 = new byte[2 *  1MB];
 allocation6 = new byte[2 *  1MB];
 allocation4 = null;
 allocation5 = null;
 allocation6 = null;
 allocation7 = new byte[2 *  1MB];
}
```

以 HandlePromotionFailure = false 的参数设置来运行的结果：

```
[GC [DefNew: 6651K->148K(9216K), 0.0078936 secs]
6651K->4244K(19456K), 0.0079192 secs] [Times: user=0.00 sys=0.02, real=0.02 secs]
[GC [DefNew: 6378K->6378K(9216K), 0.0000206 secs]
[Tenured: 4096K->4244K(10240K), 0.0042901 secs]
10474K->4244K(19456K), [Perm : 2104K->2104K(12288K)],0.0043613 secs] [Times:
user=0.00 sys=0.00, real=0.00 secs]
```

以 MaxTenuringThreshold = true 的参数设置来运行的结果：

```
[GC [DefNew: 6651K >148K(9216K), 0.0054913 secs]
6651K >4244K(19456K), 0.0055327 secs] [Times:user-0.00 sys-0.00, real 0.00 secs]
[GC [DefNew: 6378K->148K(9216K), 0.0006584 secs]
10474K->4244K(19456K), 0.0006857 secs] [Times: user-0.00 sys=0.00, real=0.00
secs]
```

本章介绍了垃圾收集的算法、几款 JDK 1.6 中提供的垃圾收集器的特点及其运作原理。通过代码实例验证了 Java 虚拟机中自动内存分配及回收的主要规则。

内存回收与垃圾收集器在很多时候都是影响系统性能和并发能力的主要因素之一，虚拟机之所以提供多种不同的收集器及大量的调节参数，是因为只有根据实际应用需求和实现方式选择最优的收集方式才能获取最好的性能。没有固定收集器和参数组合，也就没有最优的调优方法，虚拟机也没有什么必然的内存回收行为。因此学习虚拟机的内存知识，如果要到实践调优阶段，必须了解每个具体收集器的行为、优势和劣势、调节参数。

Java

第 9 章

高效手段之性能监控工具和优化部署

前面已经对虚拟机内存分配与回收技术各方面做了介绍，相信读者已经建立了一个比较完整的理论基础。理论总是作为指导实践的工具，能把这些知识投入到实际工作中才是我们的最终目的。接下来的两章，我们将从实践的角度去了解虚拟机内存管理的世界。

本章将详细讲解使用性能工具和实现优化部署的基本知识，为读者学习后面的知识打下基础。

9.1 JDK 的命令行工具

Java 开发人员肯定都知道 JDK 的 bin 目录中有“java.exe”和“javac.exe”这两个命令行工具，但并非所有程序员都了解过 JDK 的 bin 目录之中其他命令行程序的作用。每逢 JDK 更新版本时，bin 目录下命令行工具的数量和功能总会不知不觉地增加和增强。

本章笔者将介绍这些工具的其中一部分，主要介绍用于监视虚拟机和故障处理的工具。这些故障处理工具被 Sun 公司作为“礼物”附赠给 JDK 的使用者，在软件的使用说明中把它们声明为“没有技术支持并且是实验性质的”(Unsupported and Experimental)产品，但事实上这些工具都非常稳定而且功能强大，能在处理应用程序性能问题、定位故障时发挥很大的作用。

说起 JDK 的工具，读者如果比较细心的话，可能会注意到这些工具的程序体积都异常的小。其实各个工具的体积基本上都稳定在 27KB 左右。并非 JDK 开发团队刻意把它们制作得如此精炼来炫耀编程水平，而是因为这些命令行工具大多数是 jdk\lib\tools.jar 类库的一层薄包装而已，它们主要的功能代码是在 tools 类库中实现的。读者用图 9-1 和图 9-2 两张图片对比一下就可以看得很清楚。



图 9-1 bin 目录

假如读者使用的是 Linux 版本的 JDK，还会发现这些工具中很多甚至就是由 Shell 脚本直接写成的，可以用 vim 直接打开它们。JDK 开发团队选择采用 Java 代码来实现这些监控工具是有特别用意的：当应用程序部署到生产环境后，无论是直接接触物理服务器还是远程 Telnet 到服务器上都可能会受到限制。借助 tools.jar 类库里面的接口，我们可以直接在应用程序中实现功能强大的监控分析功能。

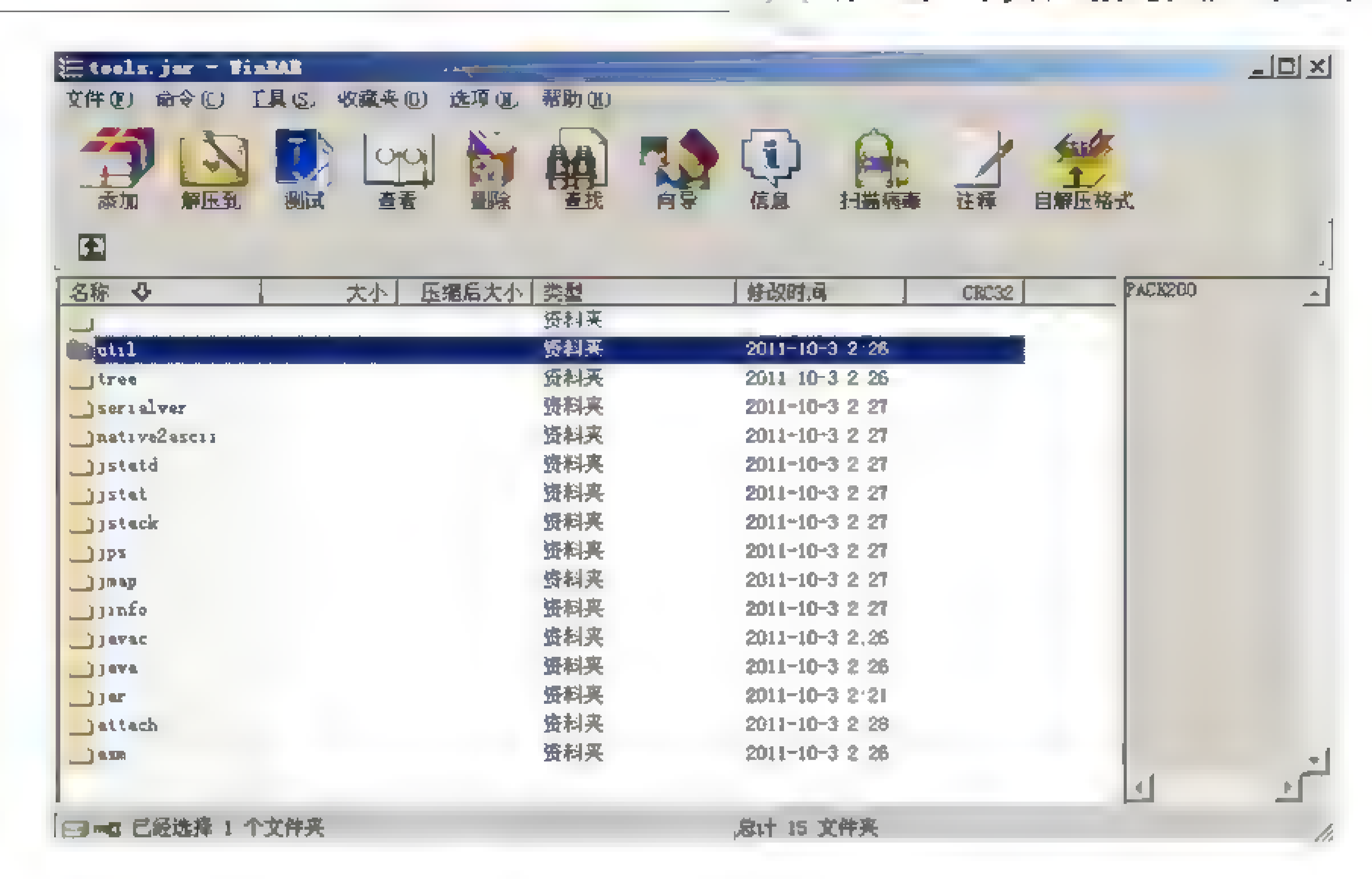


图 9-2　tools.jar 包的情况

注意：tools.jar 中的类库不属于 Java 的标准 API，如果引入这个类库，就意味着你的程序只能运行于 Sun Hotspot(或一些从 Sun 买了 JDK 的源码 License 的虚拟机，如 IBM J9、BEA JRockit)上面，或者在部署程序时需要一起部署 tools.jar。

需要特别说明的是，本章介绍的工具全部基于 Windows 平台下的 JDK 1.6 Update 21，如果 JDK 版本、操作系统不同，工具所支持的功能可能会有较大差别。大部分工具在 JDK 1.5 中就已经提供，但为了避免运行环境带来的差异和兼容性问题，建议读者使用 JDK 1.6 来验证本章介绍的内容，因为 JDK 1.6 的工具可以正常兼容运行于 JDK 1.5 的虚拟机之上的程序，反之则不一定。

如果读者在工作中需要监控运行于 JDK 1.5 的虚拟机之上的程序，在程序启动时请添加参数“-Dcom.sun.management.jmxremote”开启 JMX 管理功能，否则由于部分工具都是基于 JMX 的(包括下一节的可视化工具)，因此它们都将会无法使用，如果被监控程序运行于 JDK 1.6 的虚拟机之上，那 JMX 管理默认是开启的，虚拟机启动时无须再添加任何参数。Sun JDK 监控和故障处理工具的主要说明如下。

- jps：功能是 VM Process Status Tool，显示指定系统内所有的 HotSpot 虚拟机进程。
- jstat：功能是 JVM Statistics Monitoring Tool，用于收集 HotSpot 虚拟机各方面的运行数据。
- jinfo：功能是 Configuration Info for Java，显示虚拟机配置信息。
- jmap：功能是 Memory Map for Java，生成虚拟机的内存转储快照(heapdump 文件)。
- jhat：功能是 JVM Heap Dump Browser，用于分析 heapdump 文件，它会建立一个 HTTP/HTML 服务器，让用户可以在浏览器上查看分析结果。
- jstack：功能是 Stack Trace for Java，显示虚拟机的线程快照。

9.1.1 jps：虚拟机进程状况工具

JDK 的很多小工具的名字都参考了 Unix 命令的命名方式，例如 jps (JVM Process Status Tool)是其中的典型。除了名字像 Unix 的 ps 命令之外，它的功能也和 ps 命令类似：可以列出正在运行的虚拟机进程，并显示虚拟机执行主类(Main Class，main()函数所在的类)的名称，以及这些进程的本地虚拟机的唯一 ID(Local Virtual Machine Identifier，LVMID)。虽然功能比较单一，但它是使用频率最高的 JDK 命令行工具，因为其他的 JDK 工具大多须要输入它查询到的 LVMID 来确定要监控的是哪一个虚拟机进程。对于本地虚拟机进程来说，LVMID 与操作系统的进程 ID (Process Identifier，PID)是一致的，使用 Windows 的任务管理器或 UNIX 的 ps 命令也可以查询到虚拟机进程的 LVMID，但如果同时启动了多个虚拟机进程，无法根据进程名称定位时，那就只能依赖 jps 命令显示主类的功能才能区分了。

使用 jps 命令的格式如下：

```
jps [options] [hos tid]
```

jps 命令的功能是列出在运行的虚拟机进程，并显示虚拟机执行主类(Main?Class,?main 函数所在的类)的名称，以及这些进程的本地虚拟机的唯一 ID(LVMID,Local?Virtual?Machine?Identifier)。

jps 可以通过 RMI 协议查询开启了 RMI 服务的远程虚拟机进程状态，hostid 为 RMI 注册表中注册的主机名。jps 的其他常用选项如下。

- -q：功能是只输出 LVMID，省略主类的名称。
- -m：功能是输出虚拟机进程启动时传递给主类 main()函数的参数。
- -l：功能是输出主类的全名，如果进程执行的是 Jar 包，则输出 Jar 路径。
- -v：功能是输出虚拟机进程启动时 JVM 参数。

使用方法是进入到 java 的安装目录，位于 bin 目录下有很多的工具，其中一个名为 jps.exe 就是此工具。图 9-3 是使用 jps 命令的例子。

```
C:\Program Files\Java\jdk1.6.0_12\bin>jps
5616 RemoteFrameWork
2068 org.eclipse.equinox.launcher_1.0.1.R33x_v20080118.jar
4560 Jps
```

图 9-3　使用 jps 命令的例子

如果直接输入 jps 命令，则会显示进程 ID 和主类的名称或 jar 的名称，而且该命令还支持一些参数。例如-q 只输出 LVMID，省略主类的名称，如图 9-4 所示。



图 9-4　使用-q 选项参数

而选项-m 可以输出虚拟机进程启动时传递给主类 main()函数的参数，如图 9-5 所示。

```
C:\Program Files\Java\jdk1.6.0_12\bin>jps -m
5616 RemoteFrameWork SCHEDULER
2068 org.eclipse.equinox.launcher_1.0.1.R33x_v20080118.jar -os win32 -ws win32 -
arch x86 -showsplash -launcher E:\all\MyEclipse\eclipse\eclipse.exe -name Eclips
e --launcher.library E:\all\MyEclipse\eclipse\plugins\org.eclipse.equinox.launch
er.win32.win32.x86_1.0.3.R33x_v20080118\eclipse_1023.dll -startup E:\all\MyEclip
se\eclipse\plugins\org.eclipse.equinox.launcher_1.0.1.R33x_v20080118.jar -exitda
ta 19e0_bc -clean -vm E:\all\MyEclipse\jre\bin\javaw.exe -vmargs -Xms128m -Xmx51
2m -Duser.language=en -XX:PermSize=128M -XX:MaxPermSize=256M -jar E:\all\MyEclip
se\eclipse\plugins\org.eclipse.equinox.launcher_1.0.1.R33x_v20080118.jar
4068 Jps -m
```

图 9-5　使用-m 选项参数

而选项-l 可以输出主类的全名，如果进程执行的是 Jar 包，则输出 Jar 路径，如图 9-6 所示。

```
C:\Program Files\Java\jdk1.6.0_12\bin>jps -l
5616 com.asiainfo.boss.common.cxn.remote.RemoteFrameWork
2068 E:\all\MyEclipse\eclipse\plugins\org.eclipse.equinox.launcher_1.0.1.R33x_v2
0080118.jar
4736 sun.tools.jps.Jps
```

图 9-6　使用-l 选项参数

而-v 可以输出虚拟机进程启动时 JVM 参数，如图 9-7 所示。

```
C:\Program Files\Java\jdk1.6.0_12\bin>jps -v
5616 RemoteFrameWork -Xbootclasspath/a:E:\all\MyEclipse\myeclipse\eclipse\plugin
s\com.genuitec.eclipse.j2eedt.core_6.5.0.zmyeclipse650200806\data\libraryset\1.4
\activation.jar;E:\all\MyEclipse\myeclipse\eclipse\plugins\com.genuitec.eclipse.
j2eedt.core_6.5.0.zmyeclipse650200806\data\libraryset\1.4\javax.servlet.jar;E:\a
ll\MyEclipse\myeclipse\eclipse\plugins\com.genuitec.eclipse.j2eedt.core_6.5.0.zm
yeclipse650200806\data\libraryset\1.4\javax.servlet.jsp.jar;E:\all\MyEclipse\myc
lipse\eclipse\plugins\com.genuitec.eclipse.j2eedt.core_6.5.0.zmyeclipse65020080
6\data\libraryset\1.4\jboss-j2ee.jar;E:\all\MyEclipse\myeclipse\eclipse\plugins\
com.genuitec.eclipse.j2eedt.core_6.5.0.zmyeclipse650200806\data\libraryset\1.4\j
boss-jaxrpc.jar;E:\all\MyEclipse\myeclipse\eclipse\plugins\com.genuitec.eclipse.
j2eedt.core_6.5.0.zmyeclipse650200806\data\libraryset\1.4\jboss-jsr77.jar;E:\all
\MyEclipse\myeclipse\eclipse\plugins\com.genuitec.eclipse.j2eedt.core_6.5.0.zmye
clipse650200806\data\libraryset\1.4\jboss-saaj.jar;E:\all\MyEclipse\myeclipse\ec
lipse
2068 org.eclipse.equinox.launcher_1.0.1.R33x_v20080118.jar -Xms128m -Xmx512m -Du
ser.language=en -XX:PermSize=128M -XX:MaxPermSize=256M
5072 Jps -Denv.class.path=.; -Dapplication.home=C:\Program Files\Java\jdk1.6.0_1
2 -Xms8m
```

图 9-7　使用-v 选项参数

9.1.2　jstat：虚拟机统计信息监视工具

jstat (JVM Statistics Monitoring Tool)是用于监视虚拟机各种运行状态信息的命令行工具。它可以显示本地或远程虚拟机进程中的类装载、内存、垃圾收集、JIT 编译等运行数据，在没有 GUI 图形界面，只提供了纯文本控制台环境的服务器上，它将是运行期定位虚拟机性能问题的首选工具。

使用 jstat 命令的格式为：

```
jstat [generalOption | outputOptions vmid [interval[s|ms] [count]]]
```

对于命令格式中的 VMID 与 LVMID 需要特别说明一下：如果是本地虚拟机进程，

VMID 与 LVMID 是一致的，如果是远程虚拟机进程，那 VMID 的格式应当是：

```
jstat -gc 2764 250 20
```

jstat 命令中各个选项的具体说明如下。

1．generalOption

- -help：帮助。
- -options：打印选项。

2．outputOptions

这是一个输出选项，参数如下。

- -h n：每 n 个样本，显示 header 一次。
- -t n：在第一列显示时间戳列，时间戳时从 jvm 启动开始计算。
- -Jjvmoption：传递 jvm 选项。
- -statOption：决定统计什么信息。

(1) class：用于统计 classloader 的行为，主要选项的说明如下。

- Loaded：被读入类的数量。
- Bytes：被读入的字节数(K)。
- Unloaded：被卸载类的数量。
- Bytes：被卸载的字节数(K)。
- Time：花费在 load 和 unload 类的时间。

(2) compiler：用于统计 hotspot just-in-time 编译器的行为，主要选项的说明如下。

- Compiled：被执行的编译任务的数量。
- Failed：失败的编译任务的数量。
- Invalid：无效的编译任务的数量。
- Time：花费在执行编译任务的时间。
- FailedType：最近失败编译的编译类名。
- FailedMethod：最近失败编译的类名和方法名。

(3) gc：用于统计 gc 行为，主要选项的说明如下。

- S0C：当前 S0 的容量(KB)。
- S1C：当前 S1 的容量(KB)。
- S0U：S0 的使用(KB)。
- S1U：S1 的使用(KB)。
- EC：当前 eden 的容量(KB)。
- EU：eden 的使用 (KB)。
- OC：当前 old 的容量(KB)。
- OU：old 的使用 (KB)。
- PC ：当前 perm 的容量(KB)。
- PU：perm 的使用(KB)。

- YGC：young 代 gc 的次数。
- YGCT：young 代 gc 花费的时间。
- FGC：full gc 的次数。
- FGCT：full gc 的时间。
- GCT：垃圾收集收集的总时间。

(4) gccapacity：用于统计堆中代的容量、空间，主要选项的说明如下。

- NGCMN：年轻代的最小容量(KB)。
- NGCMX：年轻代的最大容量(KB)。
- NGC：当前年轻代的容量(KB)。
- S0C：当前 S0 的空间(KB)。
- S1C：当前 S1 的空间(KB)。
- EC：当前 eden 的空间(KB)。
- OGCMN：年老代的最小容量(KB)。
- OGCMX：年老代的最大容量(KB)。
- OGC：当前年老代的容量(KB)。
- OC：当前年老代的空间(KB)。
- PGCMN：永久代的最小容量(KB)。
- PGCMX：永久代的最大容量(KB)。
- PGC：当前永久代的容量(KB)。
- PC：当前永久代的空间(KB)。
- YGC：年轻代 gc 的次数。
- FGC：full gc 的次数。

(5) gccause：垃圾收集统计，包括最近引用垃圾收集的事件，基本同 gcutil，只是比 gcutil 多了两列。主要选项的说明如下。

- LGCC：最近垃圾回收的原因。
- GCC：当前垃圾回收的原因。

(6) gcnew：用于统计新生代的行为，主要选项的说明如下。

- S0C：当前 S0 空间(KB)。
- S1C：当前 S1 空间(KB)。
- S0U：S0 空间使用(KB)。
- S1U：S1 空间使用(KB)。
- MTT：最大的 tenuring threshold。
- DSS：希望的 Survivor 大小(KB)。
- EC：当前 eden 空间(KB)。
- EU：eden 空间使用(KB)。
- YGC：年轻代 gc 次数。
- YGCT：年轻代垃圾收集时间。

(7) gcnewcapacity：用于统计新生代的大小和空间，主要选项的说明如下。

- NGCMN：最小的年轻代的容量(KB)。
- NGCMX：最大的年轻代的容量(KB)。
- NGC：当前年轻代的容量(KB)。
- S0CMX：最大的 S0 空间(KB)。
- S0C：当前 S0 空间(KB)。
- S1CMX：最大的 S1 空间(KB)。
- S1C：当前 S1 空间(KB)。
- ECMX：最大 eden 空间(KB)。
- EC：当前 eden 空间(KB)。
- YGC：年轻代 gc 数量。
- FGC：full gc 数量。

(8) gcold：用于统计旧生代的行为，主要选项的说明如下。

- PC：当前 perm 空间 (KB)。
- PU：perm 空间使用 (KB)。
- OC：当前 old 空间 (KB)。
- OU old 空间使用 (KB)。
- YGC：年轻代 gc 次数。
- FGC：full gc 次数。
- FGCT：full gc 时间。
- GCT：垃圾收集总时间。

(9) gcoldcapacity：统计旧生代的大小和空间，主要选项的说明如下。

- OGCMN：最小年老代容量(KB)。
- OGCMX：最大年老代容量(KB)。
- OGC：当前年老代容量(KB)。
- OC：当前年老代空间(KB)。
- YGC：年轻代 gc 次数。
- FGC：full gc 次数。
- FGCT：full gc 时间。
- GCT：垃圾收集总时间。

(10) gcpermcapacity：用于统计永久代的大小和空间，主要选项的说明如下。

- PGCMN：永久代最小容量(KB)。
- PGCMX：永久代最大容量(KB)。
- PGC：当前永久代的容量(KB)。
- PC：当前永久代的空间(KB)。
- YGC：年轻代 gc 次数。
- FGC：full gc 次数。
- FGCT：full gc 时间。
- GCT：垃圾收集总时间。

(11) gcutil：实现垃圾收集统计，主要选项的说明如下。

- S0：S0 使用百分比。
- S1：S1 使用百分比。
- E：eden 使用百分比。
- O：old 使用百分比。
- P：perm 使用百分比。
- YGC：年轻代 gc 次数。
- YGCT：年轻代 gc 时间。
- FGC：full gc 次数。
- FGCT：full gc 时间。
- GCT：垃圾收集总时间。

(12) printcompilation：hotspot 编译方法统计，主要选项的说明如下。

- Compiled：被执行的编译任务的数量。
- Size：方法字节码的字节数。
- Type：编译类型。
- Method：编译方法的类名和方法名。类名使用“/”代替“.”作为空间分隔符，方法名是给出类的方法名，格式是一致于 HotSpot - XX:+PrintComplation 选项。

3. vmid

表示虚拟机标识符，格式为

```
[protocol :][//]lvmid [@hostname [:port ]/servername ]
```

4. interval：显示间隔

5. count

count 的功能是显示次数，例如每隔 5 秒显示在 127.0.0.1 机器上的 18668jvm 的 classloader 相关信息，一共显示 100 次，并且每 5 次显示一个列头，显示时间戳。每隔 5 秒显示在 127.0.0.1 机器上的 18668jvm 的 gc 统计相关信息，一共显示 100 次，并且每 5 次显示一个列头，显示时间戳。

使用 jstat 命令监测内存使用和垃圾回收统计数据：

```
$ <JDK>/bin/jstat -gcutil  [-h<lines>] <pid> <interval>
```

- jstat – gcutil：选项打印所运行应用程序进程 ID。
- <pid>：在指定抽样间隔<interval>下，堆使用及垃圾回收时间摘要，并且每<lines>行显示一次头信息。会产生如下样例输出：

```
S0     S1     E      O      P      YGC   YGCT    FGC    FGCT     GCT
0.00   0.00  24.48  46.60  90.24  142    0.530   104   28.739   29.269
0.00   0.00   2.38  51.08  90.24  144    0.536   106   29.280   29.816
0.00   0.00  36.52  51.08  90.24  144    0.536   106   29.280   29.816
0.00  26.62  36.12  51.12  90.24  145    0.538   107   29.552   30.090
```

9.1.3 jinfo：Java 配置信息工具

jinfo (Configuration Info for Java)的作用是实时查看和调整虚拟机的各项参数。使用 jps 命令的“-V”参数可以查看虚拟机启动时显式指定的参数列表，但如果想知道未被显式指定的参数的系统默认值，除了去找资料外，就只能使用 jinfo 的“-flag”选项进行查询了。如果只限于 JDK 1.6 或以上版本的话，使用 java -XX:+PrintFlagsFinal 查看参数默认值也是一个很好的选择。jinfo 还可以使用-sysprops 选项把虚拟机进程的 System.getProperties()的内容打印出来。这个命令在 JDK 1.5 版本中已经随着 Linux 版的 JDK 而发布，当时只提供了信息查询的功能。在 JDK 1.6 之后，jinfo 在 Windows 和 Linux 平台都有提供，并且加入了运行期修改参数的能力，可以使用-flag [+l-]name 或-flag name=value 修改一部分运行期可写的虚拟机参数值。JDK 1.6 中，jinfo 对于 Windows 平台的功能仍然有较大的限制，只提供了最基本的-flag 选项。

使用 jinfo 命令的格式如下：

```
jinfo[option]pid
```

例如下面的命令可以查询 CMSInitiatingOccupancyFraction 参数值。

```
C:\>jinfo -flag CMSInitiatingOccupancyFraction 1444
- xx:CMSInitiatingOccupancyFraction=85
```

9.1.4 jmap：Java 内存映像工具

通过 jmap(Memory Map for Java)命令可以生成堆转储快照(一般称为 heapdump 或 dump 文件)。如果不使用 jmap 命令，要想获取 Java 堆转储快照还有一些比较“暴力”的手段，例如用-XX：+HeapDumpOnOutOfMemoryError 参数可以让虚拟机在 OOM 异常出现之后自动生成 dump 文件，通过-XX：+HeapDumpOnCtrlBreak 参数则可以使用 Ctrl+Break 键让虚拟机生成 dump 文件，又或者在 Linux 系统下通过 Kill.3 命令发送进程退出信号“恐吓”一下虚拟机，也能拿到 dump 文件。

命令 jmap 的作用并不仅仅是为了获取 dump 文件，它还可以查询 finalize 执行队列，Java 堆和永久代的详细信息，如空间使用率、当前用的是哪种收集器等。和 jinfo 命令一样，jmap 有不少功能在 Windows 平台下都是受限的，除了生成 dump 文件的-dump 选项和用于查看每个类的实例、空间占用统计的-histo 选项所有操作系统都提供之外，其余选项都只能在 Linux/Solaris 下使用。

使用 jmap 命令的格式如下：

```
jmap[option]vmid
```

选项 option 的合法值与具体含义如下。

- -dump：功能是生成 Java 堆转储快照。格式为：

```
-dump:[live,]format=b,file=<flename>
```

  其中 liv 子参数说明是否只 dump 出存活的对象。

- -finalizerinfo：功能是显示在 F-Queue 中等待 Finalizer 线程执行 finalize 方法的对


象，只在 Linux/Solaris 平台下有效。

- -heap：功能是显示 Java 堆详细信息，如使用哪种回收器、参数配置、分代状况等，只在 Linux/Solaris 平台下有效。
- -histo：显示堆中对象统计信息，包括类、实例数量和合计容量。
- -permstat：以 ClassLoader 为统计口径显示永久代内存状态，只在 Linux/Solaris 平台下有效。
- -F：当虚拟机进程对-dump 选项没有响应时，可使用这个选项强制生成 dump 快照，只在 Linux/Solaris 平台下有效。

例如在下面的命令中，演示了使用 jmap 生成一个正在运行的 Eclipse 的 dump 快照文件的例子。

```
C:\Users\IcyFenix>jmap -dump:format=b,file=eclipse.bin 3500
Dumping heap to C:\Users\IcyFenix\eclipse.bin…
Heap dump file created
```

其中上述代码中的“3500”是通过 jps 命令查询到的 LVMID。

9.1.5 jhat：虚拟机堆转储快照分析工具

JDK 提供了 jhat (JVM Heap Analysis Tool)命令与 jmap 搭配使用，来分析 jmap 生成的堆转储快照。jhat 内置了一个微型的 HTTP/HTML 服务器，生成 dump 文件的分析结果后，可以在浏览器中查看。不在实际工作中，除非笔者手上真的没有别的工具可用，否则一般都不会去直接使用 jhat 命令来分析 dump 文件，主要原因有二：一是一般不会在部署应用程序的服务器上直接分析 dump 文件，即使可以这样做，也会尽量将 dump 文件复制到其他机器上进行分析，因为分析工作是一个耗时而且消耗硬件资源的过程，既然都要在其他机器上进行，就没必要受到命令行工具的限制了；另外一个原因是 jhat 的分析功能相对来说比较简陋，所以一般会使用 VisualVM 以及专业地用于分析 dump 文件的 Eclipse Memory Analyzer、IBM HeapAnalyzer 等工具，都能实现比 jhat 更强大、更专业的分析功能。下面的代码演示了使用 jhat 分析上一节采用 jmap 生成的 Eclipse IDE 的内存快照文件。

例如，下面的命令演示了使用 jhat 分析 dump 文件的用法。

```
C: \Users\IcyFenix>jhat eclipse.bin
Reading from eclipse.bin...
Dump file created Fri Nov 19 22:07:21 CST 2010
Snapshot read, resolving...
Resolving 1225951 Objects...
Chasing references, expect 245 dots....
Eliminating duplicate references...
Snapshot resolved.
Started HTTP server on port 7000
Server is ready.
```

当屏幕显示“Server is ready.”的提示后，用户在浏览器中输入“http://localhost:7000/”就可以看到分析结果。分析结果默认以包为单位进行分组显示，分析内存泄露问题主要会使

用到其中的“Heap Histogram”(与 jmap-histo 功能一样)与 OQL 页签的功能，前者可以找到内存中总容量最大的对象，而后者是标准的对象查询语言，使用类似 SQL 的语法对内存中的对象进行查询设计。

9.1.6 jstack：Java 堆栈跟踪工具

jstack (Stack Trace for Java)命令用于生成虚拟机当前时刻的线程快照，一般称为 threaddump 或 javacore 文件。线程快照就是当前虚拟机内每一条线程正在执行的方法堆栈的集合，生成线程快照的主要目的是定位线程出现长时间停顿的原因，如线程间死锁、死循环、请求外部资源导致的长时间等待等，都是导致线程长时间停顿的常见原因。

线程出现停顿的时候通过 jstack 来查看各个线程的调用堆栈，就可以知道没有响应的线程到底在后台做些什么事情，或者等待着什么资源。

使用 jstack 命令的格式如下：

```
jstack [option]vmid
```

option 选项的合法值与具体含义如下。

- -F：当正常输出的请求不被响应时，强制输出线程堆栈。
- -l：除堆栈外，显示关于锁的附加信息。
- -m：如果调用到本地方法的话，可以显示 C/C++的堆栈。

例如通过下面的命令，演示了使用 jstack 查看 Eclipse 线程堆栈的过程。

```
    C:\Users\IcyFenix>jstack -l 3500
    2010-11-19  23：11:26
    Full thread dump Java HotSpot(TM) 69-Bit Server VM (17.1-b03 mixed mode):
    "[ThreadPool Manager] - Idle Thread" daemon prio=6 tid=0x0000000039dd4000
    nid=0xfso  in Object.wait()  e0x000000003c96f000]
    java.lang.Thread.State: WAITING (on object monitor)
    at java.lang.Object.wait (Native Method)
    - waiting on <0x0000000016bdcc60>(a
org.eclipse.equinox.internal.util.impl.
    tpt.threadpool.Executor)
    at java .lang.Object.wait (Object.java:485)
    at org.eclipse. equinox.internal.util.impl.tpt.threadpool. Executor.
run(Executor.java:106)
     -locked< 0x0 00 00 00 016bdcc60>(a
org.aclipse.equinox.internal.util.impl.tpt.
    threadpool.Executor)
    Locked ownable synchronizers:
    - None
```

在上述命令中，“3500”是通过 jps 命令查询到的 LVMID。

从 JDK 1.5 版本开始，在 java.lang.Thread 类新增了一个 getAllStackTraces()方法用于获取虚拟机中所有线程的 StackTraceElement 对象。使用这个方法可以通过简单的几行代码就完成 jstack 的大部分功能，在实际项目中不妨调用这个方法做个管理员页面，可以随时使用浏览器来查看线程堆栈，例如，下面是查看线程状况的 JSP 代码。

```
<%@ page import "java.util.Map"%>
```

```
<html>
<head>
<title>服务器线程信息</title>
</head>
<body>
<pre>
<%
for  (Map.Entry<Thread,  StackTraceElement[]>  stackTrace:Thread.
getAllStackTraces().entrySet())  {
Thread thread=  (Thread)  stackTrace.getKey();
StackTraceElement[]  stack=  (StackTraceElement[])  stackTrace.getValue()j
if( thread.equals (Thread. currentThread()))  {
continue:
)
out .print("\n 线程: "+thread.getName()+"\n");
for (StackTraceElement element  :  stack)  {
out.print ("\t"+element+"\n");
)
)
%>
</pre>
</body>
</html>
```

9.2　JDK 的可视化工具

JDK 中除了提供大量的命令行工具外，还有两个功能强大的可视化工具：JConsole 和 VisualVM，这两个工具是 JDK 的正式成员。其中 JConsole 是在 JDK 1.5 时期就已经提供的虚拟机监控工具，而 VisualVM 在 JDK 1.6 Update7 中才首次发布，现在已经成为 Sun (Oracle)主力推动的多合一故障处理工具，并且已经从 JDK 中分离出来成为可以独立发展的开源项目。

9.2.1　JConsole：Java 监视与管理控制台

JConsole (Java Monitoring and Management Console)是一款基于 JMX 的可视化监视和管理的工具。它管理部分的功能是针对 JMX MBean 进行管理。JConsole 是一个基于 JMX 的 GUI 工具，用于连接正在运行的 JVM，不过此 JVM 需要使用可管理的模式启动。如果要把一个应用以可管理的形式启动，可以在启动是设置 com.sun.management.jmxremote。例如，启动一个可以在本地监控的 J2SE 的应用 Java2Demo，需输入以下命令：

```
JDK HOME/bin/java -Dcom.sun.management.jmxremote -jar
[b]JDK_HOME/demo/jfc/Java2D/Java2Demo.jar
```

由于 MBean 可以使用代码、中间件服务器的管理控制台或者所有符合 JMX 规范的软件进行访问，所以本节中将会着重介绍 JConsole 监视部分的功能。在命令行中输入“jconsole”后，如果弹出窗口，则说明配置可用。

1. 启用 JConsole

通过 JDK/bin 目录下的 jconsole.exe 启动 JConsole 后，然后，将自动搜索出本机运行的所有虚拟机进程，不需要用户自己再使用 jps 来查询，如图 9-8 所示。

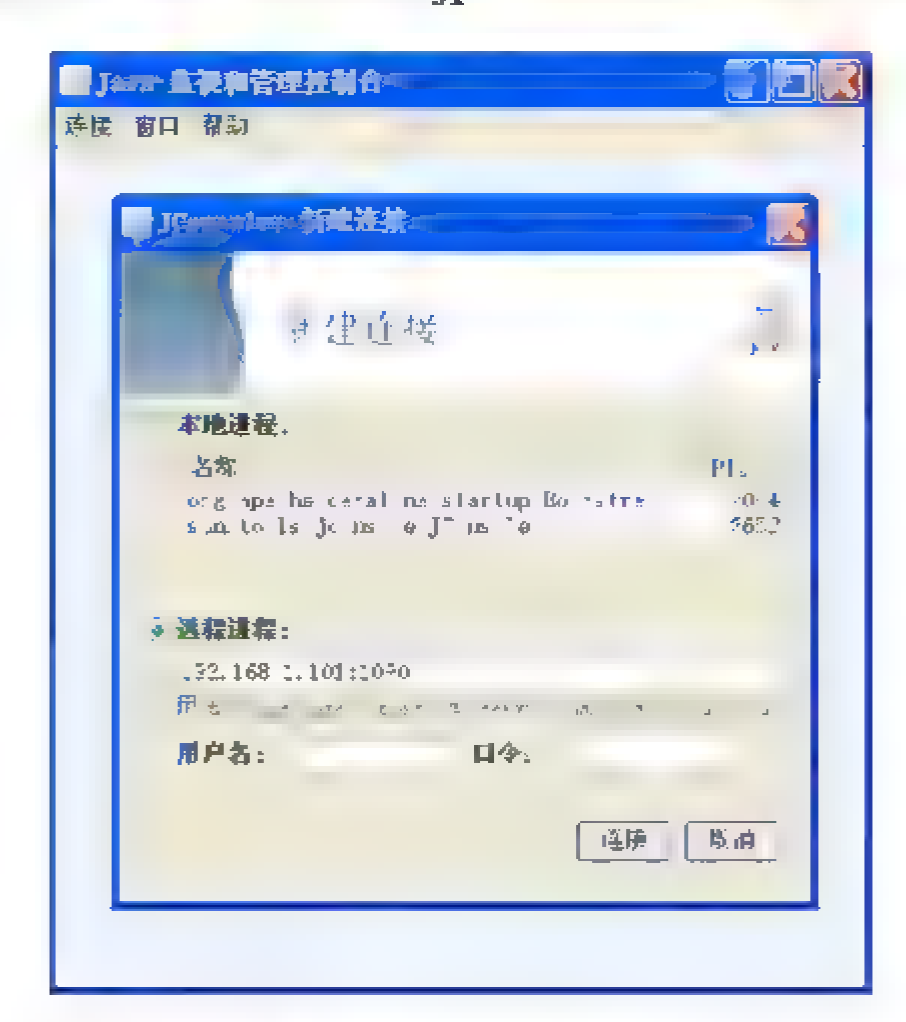


图 9-8 新建连接

在“远程进程”中输入“192.168.1.101:1090”，单击“连接”按钮就可以查看到远程 Tomcat 服务器的运行情况了，如图 9-9 所示。



图 9-9 Jconsole 主界面

在图 9-9 所示的启动界面中，各个选项卡的具体说明如下：

- 概述：有关堆内存使用情况，线程，类加载和 CPU 使用情况的综述；
- 内存：内存的详细情况，堆和其他内存；

- 线程：峰值/活动线程，另外，各个线程的明细信息，检测死锁；
- 类：监控加载和卸载的类；
- VM 摘要：有关 vm 的明细信息；
- MBean：当前 Java 程序的 MBean(如果有的话)的操作。

MBean 选项卡展示了所有以一般形式注册到 JVM 上的 MBeans。MBeans tab 允许你获取所有的平台信息，包括那些不能从其他选项卡获取到的信息。注意，其他选项卡上的一些信息也在 MBeans 这里显示。另外，你可以使用 MBeans 选项卡管理你自己的应用 Mbeans。

2. 使用 MBean 标签监控和管理 MBean

注册到 JMX 代理的平台或者应用的 MBean，可以通过 MBean 标签获取。例如下面是内存的 MBean 定义。

```
public interface MemoryMXBean {
    public MemoryUsage getHeapMemoryUsage();
    public MemoryUsage getNonHeapMemoryUsage();
    public int        getObjectPendingFinalizationCount();
    public boolean     isVerbose();
    public void       setVerbose(boolean value);
    public void       gc();
}
```

内存的 MBean 包括如下 4 个属性：

- HeapMemoryUsage：用于描述当前堆内存使用情况的只读属性。
- NonHeapMemoryUsage：用于描述当前的非堆内存的使用情况的只读属性。
- ObjectPendingFinalizationCount：用于描述有多少对象被挂起以便回收。
- Verbose：用于动态设置 GC 是否跟着详细的堆栈信息，为一个布尔变量。

内存的 MBean 支持一个操作——GC，此操作可以发送进行实时的垃圾回收请求。MBean 选项卡如图 9-10 所示。

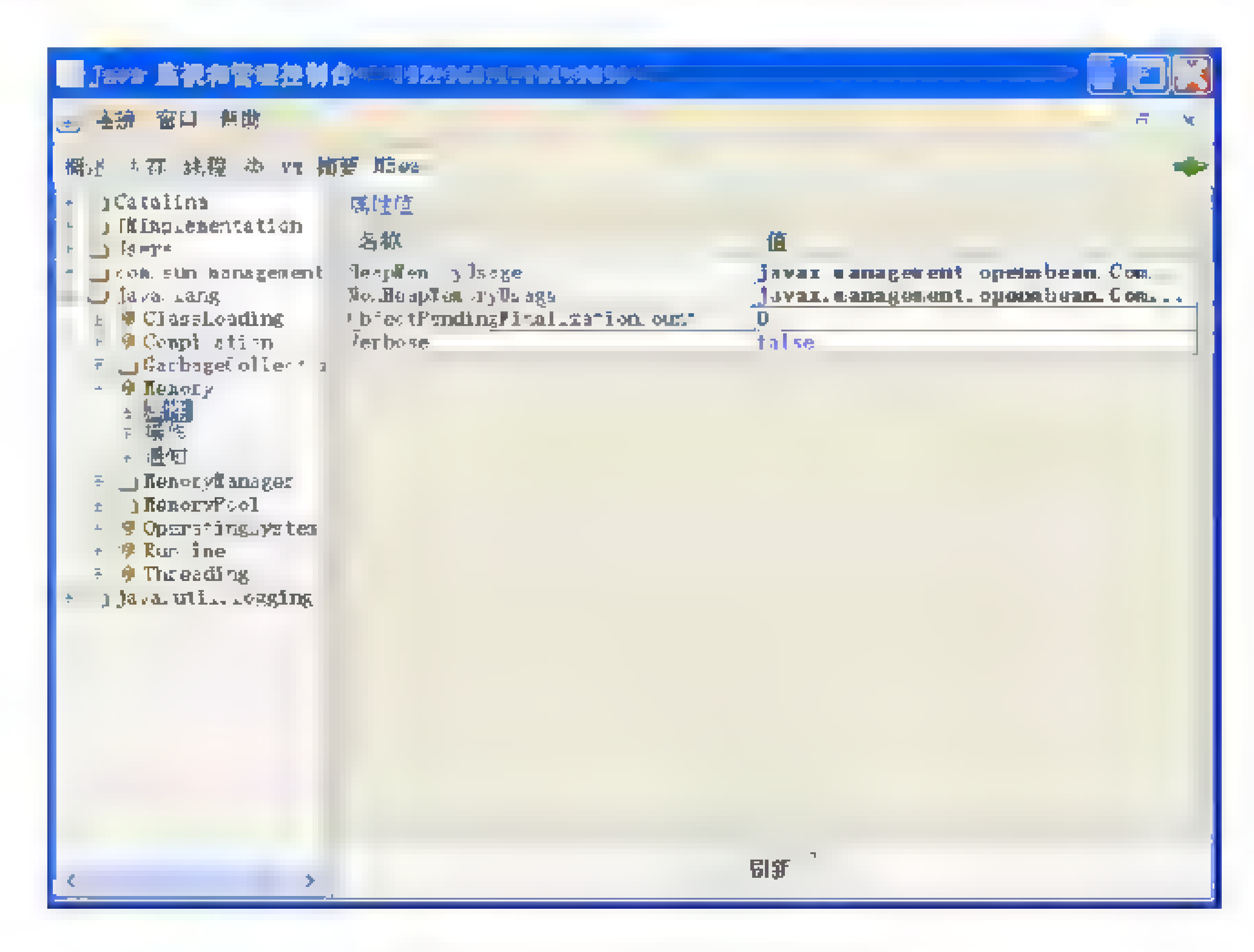


图 9-10　MBean 选项卡

左边的树形结构以名字的方式展示了所有 MBean 的列表。一个 MBean 对象的名字由一个域的名字和一串关键字属性组成。例如，JVM 的平台的 MBean 是在 java.lang 域下的一组，而日志的 MBeans 则在 java.util.logging 域下。MBean 对象的名字在 javax.management.ObjectName 规范中定义。

当我们在树中选中一个 MBean，属性、操作或者通知等一些信息会再右边显示出来，如果属性是可写的(属性被标志为蓝色)则可以进行设置。图 9-11 是 MBean 的操作界面。

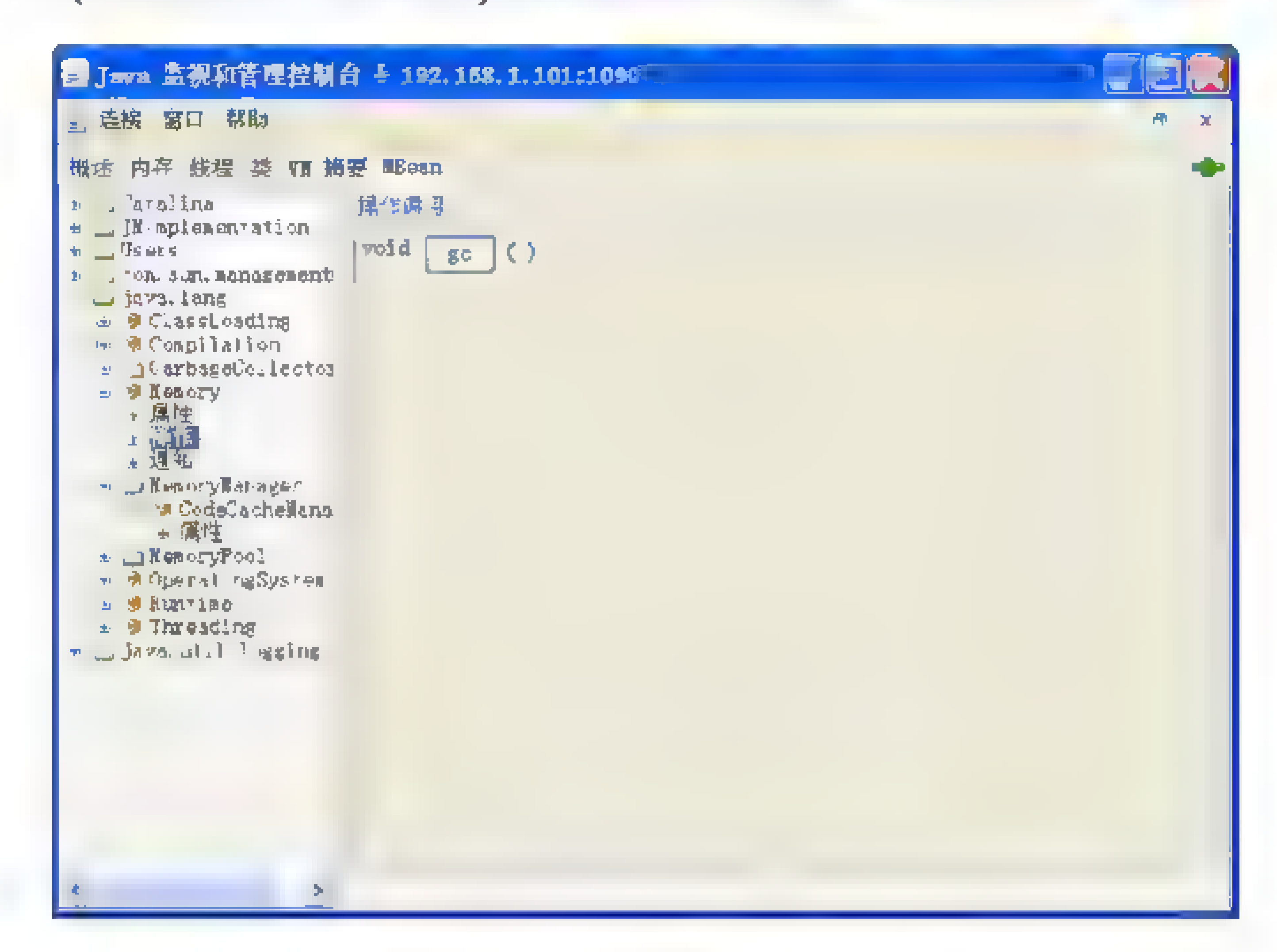


图 9-11　MBean 的操作界面

MBean 的通知界面如图 9-12 所示。

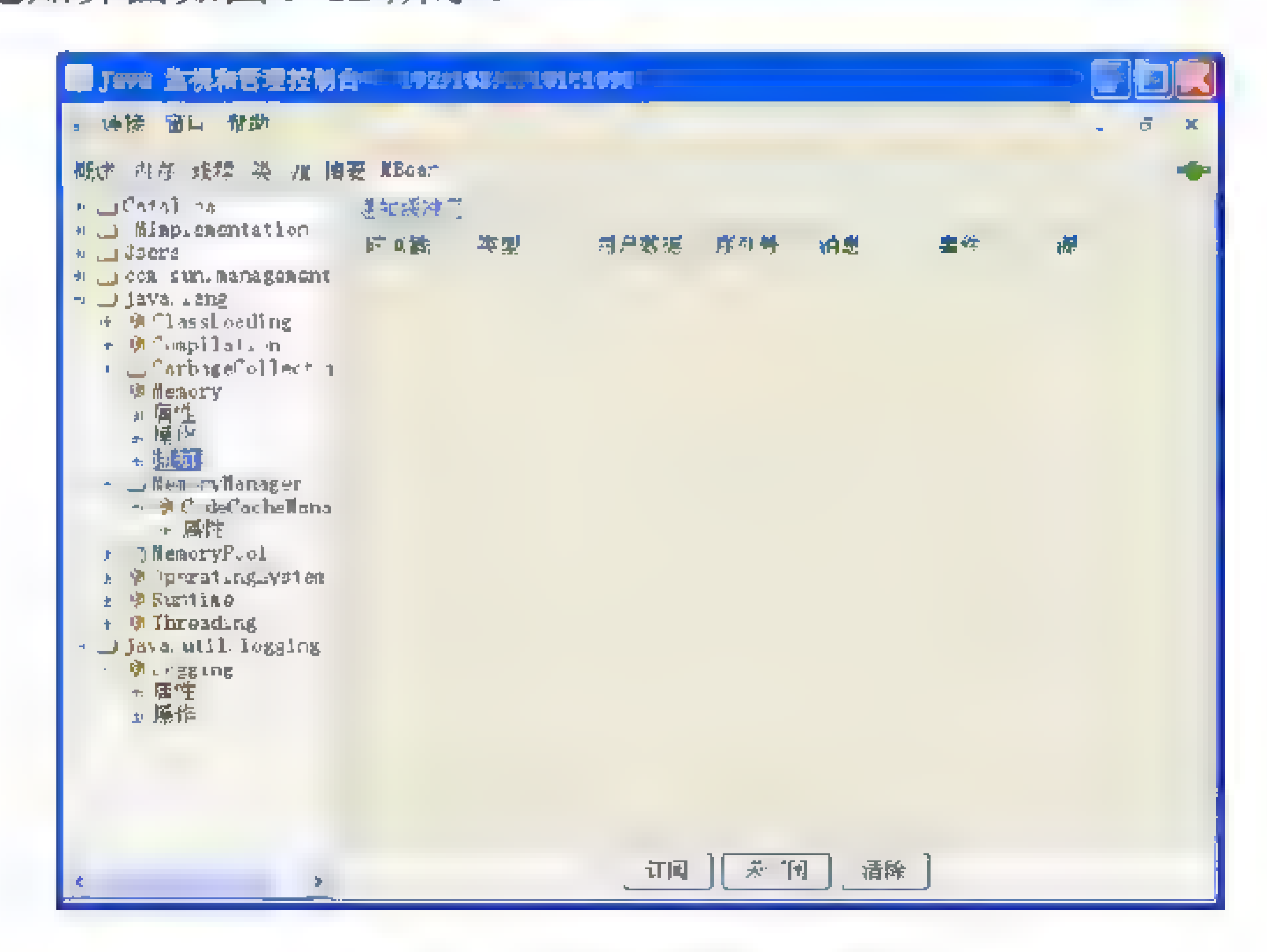


图 9-12　MBean 的通知界面

此时也可以看到由 MBean 发送出来的通知：默认情况，如果你不订阅通知的话，JConsole 不会收到 MBean 发生过来的通知。可以单击“订阅”按钮来通知进行定义，使用“未订阅”按钮可以取消订阅。

3．监控内存

内存标签页通过读取内存系统、内存池、垃圾回收的 MBean 来获取对内存消耗、内存池、垃圾回收情况的统计。监控内存的界面如图 9-13 所示。

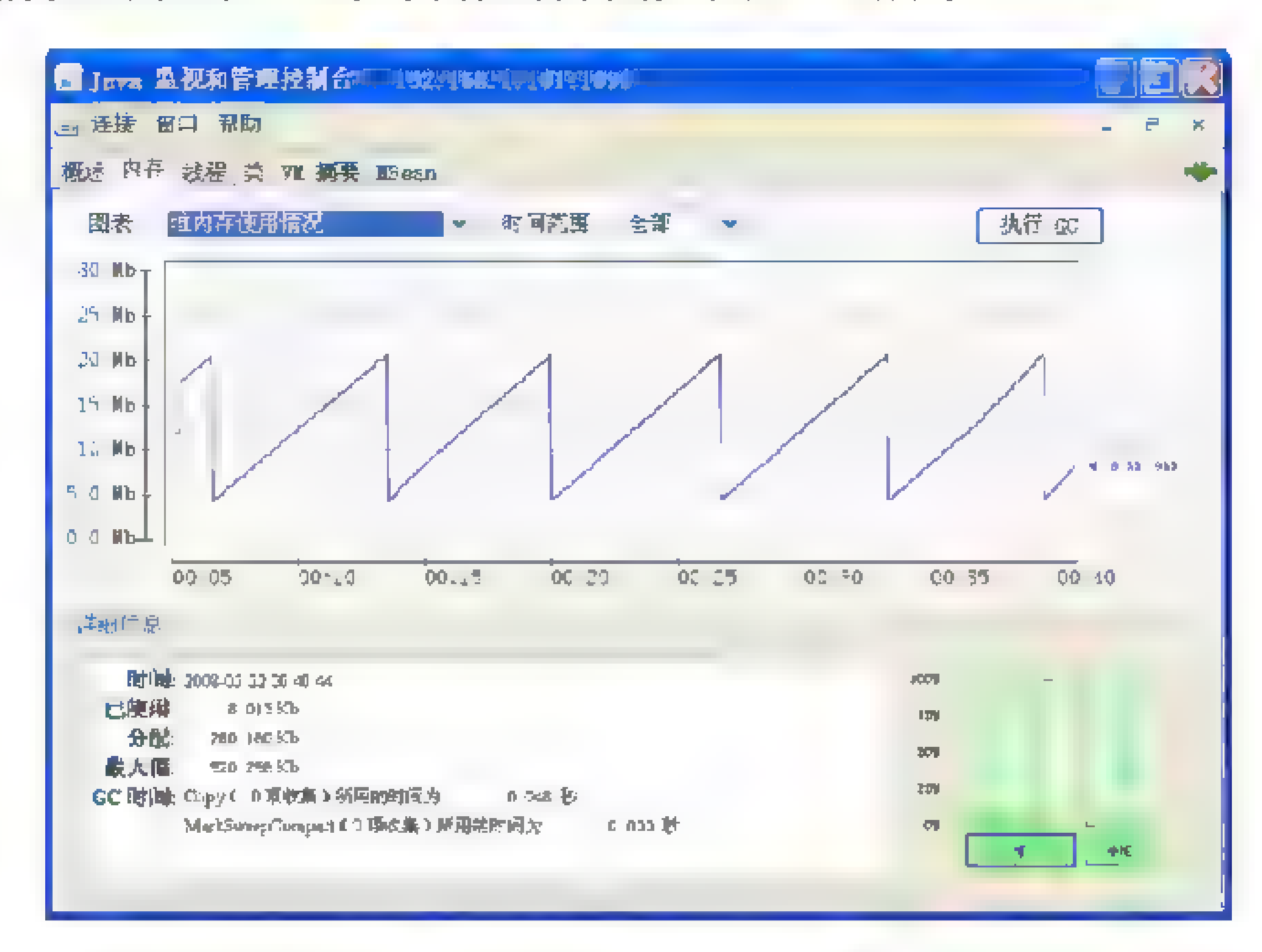


图 9-13　监控内存界面

图 9-13 展示了内存随时间变化的使用情况。有对堆的、非堆的以及特殊内存池的统计。内存池信息是否能被获取，取决于使用的 Java 虚拟机。下面列表展示了 HotSpot 虚拟机的内存池情况。

- 内存池“Eden Space”(heap)：内存最初从这个线程池分配给大部分对象。
- 内存池“Survivor Space”(heap)：用于保存在 eden space 内存池中经过垃圾回收后没有被回收的对象。
- 内存池“Tenured Generation”(heap)：用于保持已经在 survivor space 内存池中存在了一段时间的对象。
- 内存池“Perm Generation”(non-heap)：用于保存虚拟机自己的静态(refective)数据，例如类(class)和方法(method)对象。Java 虚拟机共享这些类数据。这个区域被分割为只读的和只写的。
- 内存池“Code Cache”(non-heap)：HotSpot Java 虚拟机包括一个用于编译和保存本地代码(Native Code)的内存，叫做“代码缓存区”(Code Cache)，详细信息区域列出了一些当前线程的信息，例如：
  - 已使用：当前的内存使用量。使用的内存包括所有对象(能被获取和不能被获取的)所占用的内存。
  - 分配：Java 虚拟机保证能够获取到的内存量。分配内存(Committed Memory)的量可能随时间改变。Java 虚拟机可能释放部分这里的内存给系统，相应的分配的内存这时可能少于初始化时分配的给它的量。分配量总数大于或等于已使用的内存量。
  - 最大值：内存管理系统可以使用的最大内存量。这个值可以被改变或者不做

设定。如果 JVM 试图增加使用的内存到大于分配量(Committed Memory)的情况，内存分配可能失败，即便想使用的内存量小于或者等于最大值，例如系统虚拟内存比较低时。

“内存”选项卡相当于可视化的 jstat 命令，用于监视受收集器管理的虚拟机内存(Java 堆和永久代)的变化趋势。接下来通过下面的代码来体验一下它的监视功能。运行时设置的虚拟机参数为-Xms100m -Xmx100m -XX:+UseSerialGC，这段代码的作用是以 64KB/50 毫秒的速度往 Java 堆中填充数据，一共填充 1000 次，使用 JConsole 的“内存”选项卡进行监视，观察曲线和柱状指示图的变化。

```
static class OOMObject {
    public byte[]  placeholder=new byte [64*1024];
    }
    public static void fillHeap(int num) throws InterruptedException{
    List<OOMObject> list=new ArrayList<OOMObject>();
    for (int i=0; i<num; i++)  {
    // 稍作延时，令监视曲线的变化更加明显
    Thread.sleep (50);
    list.add(new OOMObject());
    }
    System.gc(),
    }
    public static void main(String[] args) throws Exception {
    fillHeap (1000),
    }
```

其中对于内存占位符对象来说，一个 OOMObject 大约占 64K。

4. 开启/关闭虚拟机的详细跟踪

如上所述，内存系统的 MBean 定义了一个叫做 Verbose 布尔变量，让你能动态地打开或关闭详细的 GC 跟踪。详细的 GC 跟踪，将会在 JVM 启动时显示。默认的 HotSpot 的 GC 详细输出为 stdout。设置 Verbose GC 的界面如图 9-14 所示。



图 9-14　设置 Verbose GC 的界面

9.2.2　VisualVM：多合一故障处理工具

VisualVM 提供在 Java 虚拟机(Java Virutal Machine, JVM)上运行的 Java 应用程序的详细信息。在 VisualVM 的图形用户界面中，我们可以方便、快捷地查看多个 Java 应用程序的相关信息。

简单说来，VisualVM 是一种集成了多个 JDK 命令行工具的可视化工具，它能为您提供强大的分析能力。所有这些都是免费的！它囊括的命令行工具包括 jstat、JConsole、jstack、jmap 和 jinfo，这些工具与 JDK 的标准版本是一致的。 可以使用 VisualVM 生成和分析海量数据、跟踪内存泄露、监控垃圾回收器、执行内存和 CPU 分析，同时它还支持在 MBeans 上进行浏览和操作。尽管 VisualVM 自身要在 JDK 6 这个版本上运行，但是 JDK1.4 以上版本的程序它都能监控。

VisualVM 是到目前为止，随 JDK 发布的功能最强大的运行监视和故障处理程序，并且可以预见在未来一段时间内都是官方主力发展的虚拟机故障处理工具。官方在 VisualVM 的软件说明中写上了“All-in-One”的描述字样，预示着它除了运行监视、故障处理外，还提供了很多其他方面的功能。如性能分析(Profiling)，VisualVM 的性能分析功能甚至比起 JProfiler、YourKit 等专业且收费的 Profiling 工具都不会逊色多少，而且 VisualVM 的还有一个很大优点：不需要被监视的程序基于特殊 Agent 运行，因此它对应用程序的实际性能的影响很小，使得它可以直接应用在生产环境中。这个优点是 JProfiler、YourKit 等工具无法与之媲美的。

1. 安装 VisualVM

(1) 从 VisualVM 项目页下载 VisualVM 安装程序。
(2) 将 VisualVM 安装程序解压缩到本地系统。
(3) 导航至 VisualVM 安装目录的 bin 目录，然后启动应用程序。

2. 使用“应用程序”窗口

在启动应用程序后会打开 VisualVM 的主窗口。在默认情况下，“应用程序”窗口显示在主窗口的左窗格中。在“应用程序”窗口中，可以快速查看本地和远程 JVM 上运行的 Java 应用程序，如图 9-15 所示。



图 9-15　“应用程序”窗口

“应用程序”窗口是查看特定应用程序详细信息的主入口点。右键单击应用程序节点

将打开快捷菜单，从中可以选择是打开主应用程序标签，还是生成线程 dump 或堆 dump。

3. 浏览堆 Dump

VisualVM 有一个可视化窗口，通过该窗口可以轻松浏览堆 dump。您可以装入现有堆 dump，或为本地运行的应用程序生成堆快照。

要生成本地应用程序的堆 dump，可以执行下列任一操作：

(1) 在“应用程序”窗口中右键单击应用程序节点，然后选择“堆 Dump”。

(2) 在“应用程序”窗口中双击应用程序节点以打开应用程序选项卡，然后在“监视”选项卡中单击“堆 Dump”。

要打开保存的堆 dump，请从主菜单中选择“文件”→“装入”命令，然后找到保存的堆 dump，如图 9-16 所示。浏览打开的堆 dump 的流程如下：

(1) 单击“堆 Dump”工具栏中的“类”，以查看活动类和对应实例的列表。

(2) 双击某个类名打开“实例”视图，以查看实例列表。

(3) 从列表中选择某个实例，以查看对该实例的引用。

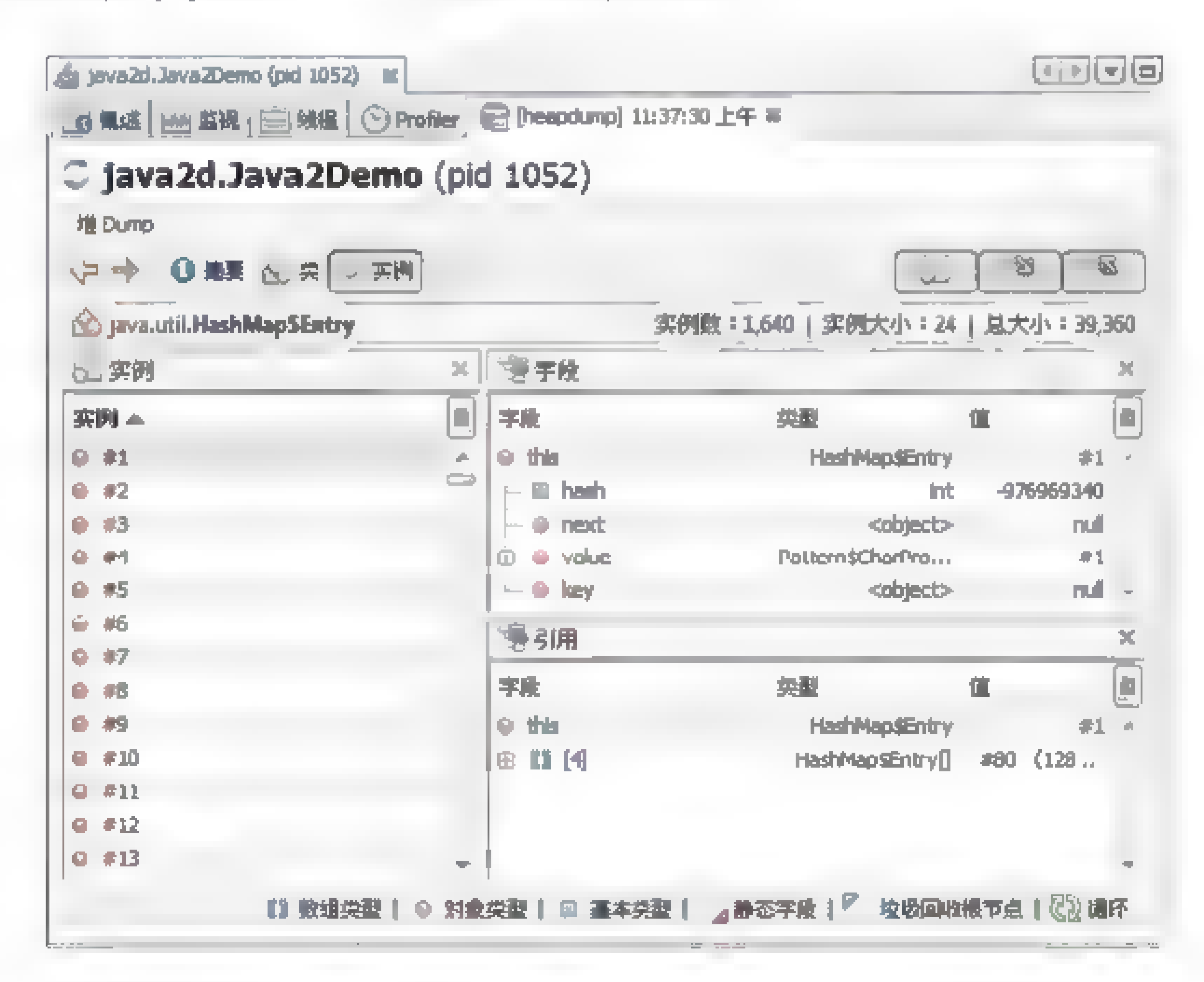


图 9-16　浏览打开的堆 dump

在生成堆 dump 后，VisualVM 将在新标签中打开该堆 dump，并在“应用程序”窗口中的应用程序节点下为该堆 dump 创建一个节点。要保存生成的堆 dump，请右键单击该堆 dump 节点，然后选择“另存为”命令。如果没有明确保存生成的堆 dump，则在应用程序关闭时将删除该 dump。

4. 分析对应用程序进行性能

VisualVM 包括一个 Profiler，可以使用它对本地 JVM 上运行的应用程序进行性能分析。您可以在应用程序标签的 Profiler 选项卡中访问性能分析控件。通过该 Profiler，可以分析本地应用程序的内存使用情况和 CPU 性能。

(1) 启动本地 Java 应用程序。(使用 -Xshare:off 参数启动该应用程序。)

(2) 在“应用程序”窗口的“本地”节点下，右键单击该应用程序节点，然后选择“打开”命令以打开该应用程序标签。

(3) 在该应用程序窗口中单击 Profiler 标签。

(4) 在 Profiler 选项卡中单击“内存”或 CPU 按钮，在选择性能分析任务后，VisualVM 将在 Profiler 选项卡中显示性能分析数据。

性能分析界面如图 9-17 所示。

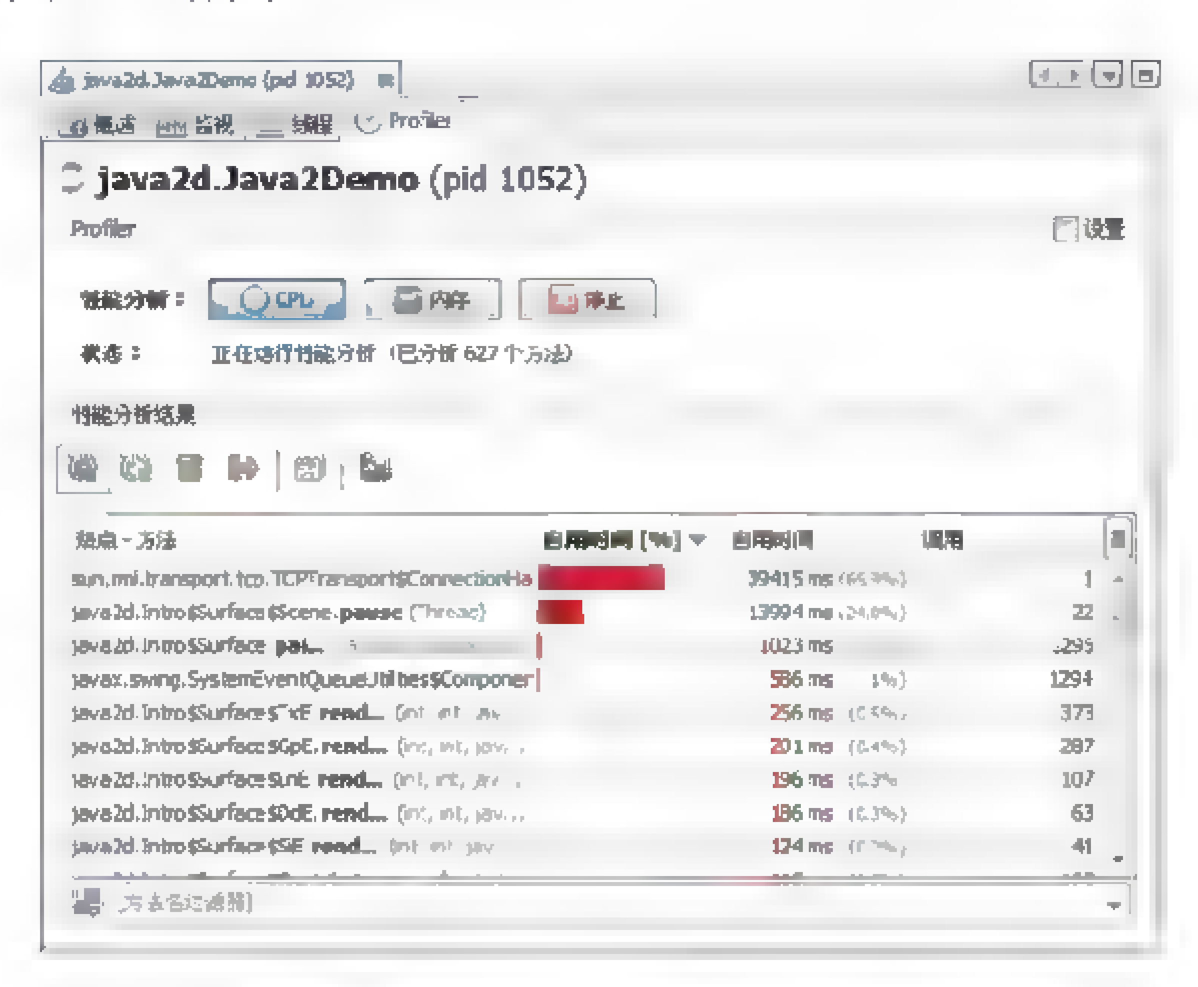


图 9-17　性能分析界面

注意：要对 JDK 6 上运行的应用程序进行性能分析，需要关闭该应用程序的类共享，否则该应用程序可能会崩溃。要关闭类共享，请使用 -Xshare:off 参数启动应用程序。

5. 连接到远程主机

通过 VisualVM 可以轻松监视远程主机上运行的应用程序，并查看有关远程系统的常规数据。要查看远程主机上应用程序的相关信息，必须首先连接到远程主机。已连接的远程主机将列在“应用程序”窗口的“远程”节点下。展开远程主机节点可查看远程主机上运行的应用程序。

要从远程应用程序中检索数据，需要在远程 JVM 上运行 jstatd 实用程序。无法对远程主机上运行的应用程序进行性能分析。

(1) 右键单击“应用程序”窗口中的“远程”节点，然后选择“添加远程主机”。

(2) 在“添加远程主机”对话框中，输入远程计算机的主机名或 IP 地址。(可选)输入远程主机的显示名称。此名称将显示在“应用程序”窗口中。如果没有输入显示名称，则在“应用程序”窗口中使用主机名标识远程主机。

(3) 单击“确定”按钮。

单击“确定”按钮后，将在“远程”节点下显示远程主机的节点。展开远程主机节点可查看远程主机上运行的 Java 应用程序。

我们可以双击远程“应用程序”的名称，在 VisualVM 中打开该“应用程序”标签。

“应用程序”界面如图 9-18 所示。



图 9-18 应用程序界面

6．安装 VisualVM 插件

通过安装 VisualVM 更新中心提供的插件，可以向 VisualVM 添加功能。例如，安装 VisualVM-MBeans 插件可以向“应用程序”窗口中添加 MBean 选项卡，通过此选项卡，可以在 VisualVM 内监视和管理 MBean。

安装 VisualVM 插件的流程如下。

(1) 从主菜单中选择“工具”→“插件”。

(2) 在“可用插件”标签中，选中该插件的“安装”复选框，单击“安装”。

(3) 逐步完成插件安装程序。

插件界面如图 9-19 所示。在此界面中显示了选中 VisualVM-MBeans 插件的“插件”管理器。

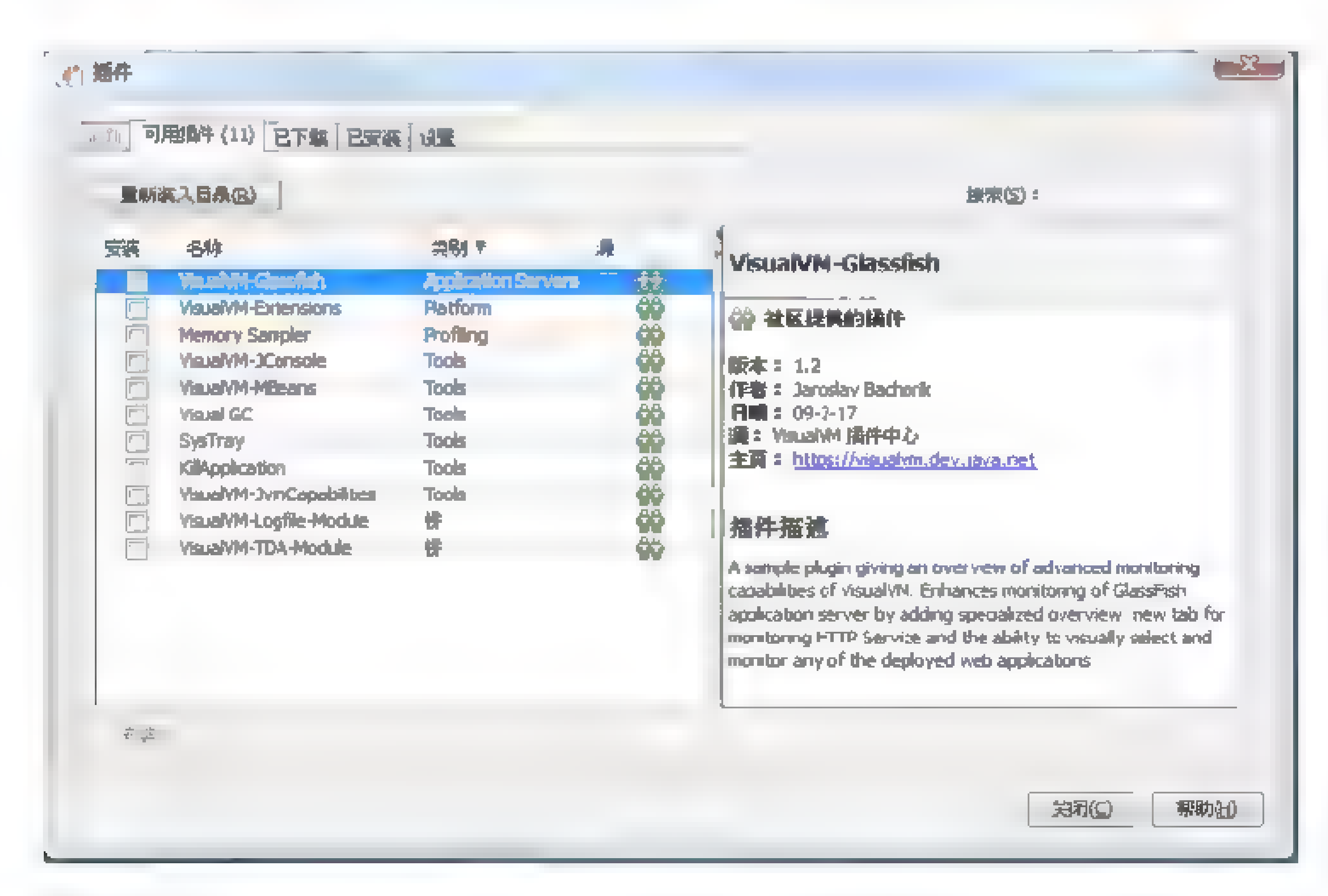


图 9-19 插件界面

本节只是简单介绍了 VisualVM 的某些功能，VisualVM 旨在提供一个直观的可视界面，使我们可以轻松浏览有关本地和远程 JVM 上运行的 Java 应用程序的信息。有关使用 VisualVM 功能的更多详细信息，读者可以从 http://visualvm.java.net/docindex.html 获取。

第 10 章

JVM 参数分析和调优实战

前面介绍了处理 Java 虚拟机内存问题的知识与工具，本章将讲解常用 JVM 参数的基本知识，并与读者分享几个比较有代表性的实际案例。考虑到虚拟机故障处理和调优主要面向各类服务端应用，而大部分 Java 程序员较少有机会直接接触生产环境的服务器，因此本章还准备了一个所有开发人员都能够进行“亲身实战”的练习，希望通过实践能使读者获得故障处理和调优的经验。

10.1 捕鱼工具选择——JVM 参数

本节讲解的 JVM 参数和内存管理有关，例如和可分配内存、非堆内存和堆内存等有密切联系。

10.1.1 通用的 JVM 参数

通用的 JVM 参数如下。

(1) -server：如果不配置该参数，JVM 会根据应用服务器硬件配置自动选择不同模式，server 模式启动比较慢，但是运行期速度得到了优化，适合于服务器端运行的 JVM。

(2) -client：启动比较快，但是运行期响应没有 server 模式的优化，适合于个人 PC 的服务开发和测试。

(3) -Xmx：设置 java heap 的最大值，默认是机器物理内存的 1/4。这个值决定了最多可用的 Java 堆内存：分配过少就会在应用中需要大量内存作缓存或者临时对象时出现 OOM(Out Of Memory)的问题；如果分配过大，那么就会因 PermSize 过小而引起的另外一种 Out Of Memory。所以如何配置还是根据运行过程中的分析和计算来确定，如果不能确定还是采用默认的配置。

(4) -Xms：设置 Java 堆初始化时的大小，默认情况是机器物理内存的 1/64。这个主要是根据应用启动时消耗的资源决定，分配少了申请起来会降低运行速度，分配多了也浪费。

(5) -XX:PermSize：初始化永久内存区域大小。永久内存区域全称是 Permanent Generation space，是指内存的永久保存区域，程序运行期不对 PermGen space 进行清理，所以如果你的 APP 会 LOAD 很多 CLASS 的话，就很可能出现 PermGen space 错误。这种错误常见在 web 服务器对 JSP 进行 pre compile 的时候。 如果你的 WEB APP 下用了大量的第三方 jar，其大小超过了 jvm 默认的 PermSize 大小(4M)那么就会产生此错误信息了。

(6) -XX:MaxPermSize：设置永久内存区域最大大小。

(7) -Xmn：直接设置青年代大小。整个 JVM 可用内存大小=青年代大小 + 老年代大小 + 持久代大小。持久代一般固定大小为 64m，所以增大年轻代后，将会减小老年代大小。此值对系统性能影响较大，Sun 官方推荐配置为整个堆的 3/8。按照 Sun 的官方设置比例，则上面的例子中年轻代的大小应该为 2048*3/8=768M。

(8) -XX:NewRatio：控制默认的 Young 代的大小，例如，设置-XX:NewRatio=3 意味着 Young 代和老年代的比率是 1∶3。换句话说，Eden 和 Survivor 空间总和是整个堆大小的 1/4。

正如图 10-1 中所示的实际设置，-XX:NewRatio=2，-Xmx=2048，则年轻代和老年代的分配比例为 1∶2，即年轻代的大小为 682M，而老年代的大小为 1365M。查看实际系统的 JVM 监控结果为：

- 内存池名称：Tenured Gen。

- Java 虚拟机最初向操作系统请求的内存量：3 538 944 字节。
- Java 虚拟机实际能从操作系统获得的内存量：1 431 699 456 字节。
- Java 虚拟机可从操作系统获得的最大内存量：1 431 699 456 字节。请注意，并不一定能获得该内存量。
- Java 虚拟机此时使用的内存量：1 408 650 472 字节，即 1 408 650 472 字节=1365M，这证明了上面的计算是正确的。



图 10-1　参数设置界面

(9) -XX:SurvivorRatio：设置年轻代中 Eden 区与 Survivor 区的大小比值。设置为 4，则两个 Survivor 区与一个 Eden 区的比值为 2∶4，一个 Survivor 区占整个年轻代的 1/6。越大的 survivor 空间可以允许短期对象尽量在年青代消亡；如果 Survivor 空间太小，Copying 收集将直接将其转移到老年代中，这将加快老年代的空间使用速度，引发频繁的完全垃圾回收。例如在图 10-2 中，SurvivorRatio 的值设为 3，Xmn 为 768M，则每个 Survivor 空间的大小为 768M/5=153.6M。

```
<jvm-options>-Xmn768M</jvm-options>
<jvm-options>-XX:SurvivorRatio=3</jvm-options>
```

图 10-2　设置-XX:SurvivorRatio 的值

(10) -XX:NewSize：为了实现更好的性能，您应该对包含短期存活对象的池的大小进行设置，以使该池中的对象的存活时间不会超过一个垃圾回收循环。新生成的池的大小由 NewSize 和 MaxNewSize 参数确定。通过这个选项可以设置 Java 新对象生产堆内存。在通常情况下这个选项的数值为 1024 的整数倍，并且大于 1MB。这个值的取值规则为，一般情况下这个值-XX:NewSize 是最大堆内存(Maximum Heap Size)的 1/4。增加这个选项值

的大小是为了增大较大数量的短生命周期对象。增加 Java 新对象生产堆内存相当于增加了处理器的数目。并且可以并行地分配内存，但是请注意内存的垃圾回收却是不可以并行处理的。作用跟-XX:NewRatio 相似，-XX:NewRatio 用来设置比例，而-XX:NewSize 是设置精确的数值。

(11) -XX:MaxNewSize：通过这个选项可以设置最大 Java 新对象生产堆内存。通常情况下这个选项的数值为 1024 的整数倍并且大于 1MB，其功用与上面的设置新对象生产堆内存-XX:NewSize 相同。一般要将 NewSize 和 MaxNewSize 设成一致。

(12) -XX:MaxTenuringThreshold：设置垃圾最大年龄。如果设置为 0 的话，则年轻代对象不经过 Survivor 区，直接进入老年代。对于老年代比较多的应用，可以提高效率。如果将此值设置为一个较大值，则年轻代对象会在 Survivor 区进行多次复制，这样可以增加对象在年轻代的存活时间，增加在年轻代即被回收的概率。例如在图 10-3 中，-XX:MaxTenuringThreshold 参数被设置成 5，表示对象会在 Survivor 区进行 5 次复制后如果还没有被回收才会被复制到老年代。

```
<jvm-options>-XX:MaxTenuringThreshold=5</jvm-options>
```

图 10-3　设置垃圾最大年龄

(13) -XX:GCTimeRatio：设置垃圾回收时间占程序运行时间的百分比。该参数设置为 n 的话，则垃圾回收时间占程序运行时间百分比的公式为 1/(1+n)，如果 n=19 表示 java 可以用 5%的时间来做垃圾回收，1/(1+19)=1/20=5%。

(14) -XX:TargetsurvivorRatio：该值是一个百分比，控制允许使用的救助空间的比例，默认值是 50。该参数设置较大的话可提高对 survivor 空间的使用率。当较大的堆栈使用较低的 SurvivorRatio 时，应增加该值到 80～90，以更好利用救助空间。

(15) -Xss：设置每个线程的堆栈大小，根据应用的线程所需内存大小进行调整，在相同物理内存下，减小这个值能生成更多的线程。但是操作系统对一个进程内的线程数还是有限制的，不能无限生成，经验值在 3000～5000 之间。当这个选项被设置的较大(>2MB)时将会在很大程度上降低系统的性能。因此在设置这个值时应该格外小心，调整后要注意观察系统的性能，不断调整以期达到最优。

JDK5.0 以后每个线程堆栈大小为 1M，以前每个线程堆栈大小为 256K。

(16) -Xnoclassgc：这个选项用来取消系统对特定类的垃圾回收。它可以防止当这个类的所有引用丢失之后，这个类仍被引用时不会再一次被重新装载，因此这个选项将增大系统堆内存的空间。禁用类垃圾回收，性能会高一点。

10.1.2　串行收集器参数

JVM 的串行收集器参数只有一个，即-XX:+UseSerialGC，功能是设置串行收集器。

10.1.3　并行收集器参数

(1) -XX:+UseParallelGC:：选择垃圾收集器为并行收集器，此配置仅对年轻代有效，

即上述配置下，年轻代使用并行收集，而老年代仍旧使用串行收集。采用了多线程并行管理和回收垃圾对象，提高了回收效率，提高了服务器的吞吐量，适合于多处理器的服务器。

(2) -XX:ParallelGCThreads：配置并行收集器的线程数，即：同时多少个线程一起进行垃圾回收。此值最好配置与处理器数目相等。

(3) -XX:+UseParallelOldGC：采用对于老年代并发收集的策略，可以提高收集效率。JDK 6.0 支持对老年代并行收集。

(4) -XX:MaxGCPauseMillis：设置每次年轻代并行收集最大暂停时间，如果无法满足此时间，JVM 会自动调整年轻代大小以满足此值。

(5) -XX:+UseAdaptiveSizePolicy：设置此选项后，并行收集器会自动选择年轻代区大小和相应的 Survivor 区比例，以达到目标系统规定的最低响应时间或者收集频率等，此值建议使用并行收集器时，一直打开。

10.1.4　并发收集器参数

(1) -XX:+UseConcMarkSweepGC：指定在老年代使用 concurrent cmark sweep gc。gc thread 和 app thread 并行(在 init-mark 和 remark 时 pause app thread)。app pause 时间较短，适合交互性强的系统，如 web server。它可以并发执行收集操作，降低应用停止时间，同时它也是并行处理模式，可以有效地利用多处理器的系统的多进程处理。

(2) -XX:+UseParNewGC 指定在 New Generation 使用 parallel collector，是 UseParallelGC 的 gc 的升级版本，有更好的性能或者优点，可以和 CMS gc 一起使用。

(3) -XX:+UseCMSCompactAtFullCollection：打开对老年代的压缩。可能会影响性能，但是可以消除碎片，在 FULL GC 的时候，压缩内存，CMS 是不会移动内存的，因此，这个非常容易产生碎片，导致内存不够用，因此内存的压缩这个时候就会被启用。增加这个参数是个好习惯。

(4) -XX:+CMSIncrementalMode：设置为增量模式，适用于单 CPU 情况。

(5) -XX:CMSFullGCsBeforeCompaction：由于并发收集器不对内存空间进行压缩、整理，所以运行一段时间以后会产生“碎片”，使得运行效率降低。此值设置运行多少次 GC 以后对内存空间进行压缩、整理。

(6) -XX:+CMSClassUnloadingEnabled：使 CMS 收集持久代的类，而不是 fullgc。

(7) -XX:+CMSPermGenSweepingEnabled：使 CMS 收集持久代的类，而不是 fullgc。

(8) -XX:-CMSParallelRemarkEnabled：在使用 UseParNewGC 的情况下，尽量减少 mark 的时间。

(9) -XX:CMSInitiatingOccupancyFraction：说明老年代到百分之多少满的时候开始执行对老年代的并发垃圾回收(CMS)，这个参数设置有很大技巧，基本上满足公式：

```
(Xmx-Xmn)*(100-CMSInitiatingOccupancyFraction)/100>=Xmn
```

时就不会出现 promotion failed。在我的应用中 Xmx 是 6000，Xmn 是 500，那么 Xmx-Xmn 是 5500 兆，也就是老年代有 5500 兆，CMSInitiatingOccupancyFraction=90 说明老年代到 90%满的时候开始执行对老年代的并发垃圾回收(CMS)，这时还剩 10%的空间是

5500*10%=550 兆，所以即使 Xmn(也就是年轻代共 500 兆)里所有对象都搬到老年代里，550 兆的空间也足够了，所以只要满足上面的公式，就不会出现垃圾回收时的 promotion failed；如果按照“Xmx=2048,Xmn=768”的比例计算，则 CMSInitiatingOccupancyFraction 的值不能超过 40，否则就容易出现垃圾回收时的 promotion failed。

(10) -XX:+UseCMSInitiatingOccupancyOnly：设置只有在老年代在使用了初始化的比例后 concurrent collector 启动收集。

(11) -XX:SoftRefLRUPolicyMSPerMB：相对于客户端模式的虚拟机(-client 选项)，当使用服务器模式的虚拟机时(-server 选项)，对于软引用(soft reference)的清理力度要稍微差一些。可以通过增大-XX:SoftRefLRUPolicyMSPerMB 来降低收集频率。默认值是 1000，也就是说每秒一兆字节。Soft reference 在虚拟机中比在客户集中存活的更长一些。其清除频率可以用命令行参数 -XX:SoftRefLRUPolicyMSPerMB=<N> 来控制，这可以指定每兆堆空闲空间的 soft reference 保持存活(一旦它不强可达了)的毫秒数，这意味着每兆堆中的空闲空间中的 soft reference 会(在最后一个强引用被回收之后)存活 1 秒钟。注意，这是一个近似的值，因为 soft reference 只会在垃圾回收时才会被清除，而垃圾回收并不总在发生。

(12) -XX:LargePageSizeInBytes：内存页的大小，不可设置过大，会影响 Perm 的大小。

(13) -XX:+UseFastAccessorMethods：原始类型的快速优化，“get,set”方法转成本地代码。

(14) -XX:+DisableExplicitGC：禁止 java 程序中的 full gc，如 System.gc() 的调用。最好加上防止程序在代码里误用了，对性能造成冲击。

(15) -XX:+AggressiveHeap：试图使用大量的物理内存长时间大内存使用的优化，能检查计算资源(内存，处理器数量)，至少需要 256MB 内存。

(16) -XX:+AggressiveOpts：加快编译。

(17) -XX:+UseBiasedLocking：锁机制的性能改善。

10.2 测试调优

本节将测试被测系统使用不同的垃圾回收方案时的性能表现，了解各种 JVM 参数在性能调优时的实际效果，将挑选出的最优方案进行 8 小时压力测试并记录测试结果。

10.2.1 测试环境准备

被测程序运行的软硬件环境如下：

- D630 4G 内存＋T7250 双核 CPU＋160G 硬盘；
- 操作系统：Windows XP SP3；
- IP：11.55.15.51。

被测程序名称：

XXX 局采购管理系统 V1.1 版。

程序部署环境：

- Tomcat 6.0.18 for Windows；
- Sun JDK 1.6.13 for Windows；
- Oracle10g for Windows(单独运行在另外一台 640M 笔记本上)。

性能测试工具运行的软硬件环境：

- 操作系统：Windows7；
- 浏览器版本：IE 8；
- IP 地址：11.55.15.141；
- 性能测试工具：Loadrunner 9.10。

JVM 监控工具：

使用 Jconsole 进行图形化监控。

确定 JVM 在被测系统的机器上最大可用内存：

通过在命令行下用 java -XmxXXXXM -version 命令反复测试发现在 11.55.15.51 机器上 JVM 能使用的最大内存为 1592M，如图 10-4 所示。

```
C:\Users\Administrator>java -Xmx1592M -version
java version "1.6.0_18"
Java(TM) SE Runtime Environment (build 1.6.0_18-b07)
Java HotSpot(TM) Client VM (build 16.0-b13, mixed mode)

C:\Users\Administrator>java -Xmx1593M -version
Error occurred during initialization of VM
Could not reserve enough space for object heap
Could not create the Java virtual machine.
```

图 10-4　测试最大内存

10.2.2　录制测试脚本

在录制前需要修改文件 checkcode.java，将随机生成的校验码改成一个固定的校验码方便脚本的自动运行，然后将编译好的 checkcode.class 文件替换发布包中的 class 文件。

选择典型业务操作进行脚本录制，每个系统的典型业务操作都会不同，需要经过分析统计，选择用户操作频率最高的部分。经过分析后确定的脚本内容为：录制系统登录操作并在登录成功后的主界面上选取一段文字作为验证点。

启动 VuGen 程序按脚本定义内容进行录制并调试脚本，保证脚本能正常运行。

10.2.3　定义测试场景

- 虚拟用户数：30。
- 持续运行时间：8 小时。
- 虚拟用户加载和卸载方式：同时。
- 性能监控指标：响应时间、吞吐量、成功交易数。

10.2.4 执行初步性能测试

使用系统默认的参数执行测试，并记录响应时间、吞吐量已经成功交易数等数据，同时监控 JVM 的使用情况。

10.2.5 选择调优方案

不同垃圾回收方法测试数据如表 10-1 所示。

表 10-1 不同垃圾回收方法测试数据

Id	NewRatio	SurviorRatio	TransResponse Time	Throughput	Passed Transactions
1	2	25	3.139s	3016230.514	7528
2	1	25	3.161s	2975581.301	7452
3	3	25	2.814s	3334717.818	8383
4	4	25	2.659s	3505592.450	8846
5	5	25	2.860s	3270596.069	8232
6	4	15	2.499s	3765121.986	9426
7	4	5	1.986s	4750776.581	11843
8	4	4	1.968s	4825608.161	11947
9	4	3	2.507s	3770420.243	9388
10	-XX:TargetSurvivorRatio=90		1.924	4945053.874	12216
11	-Xmx1024M		1.903	4974137.908	12360

使用并发收集模式，运行 10 分钟后的对内存使用情况如图 10-5 所示。

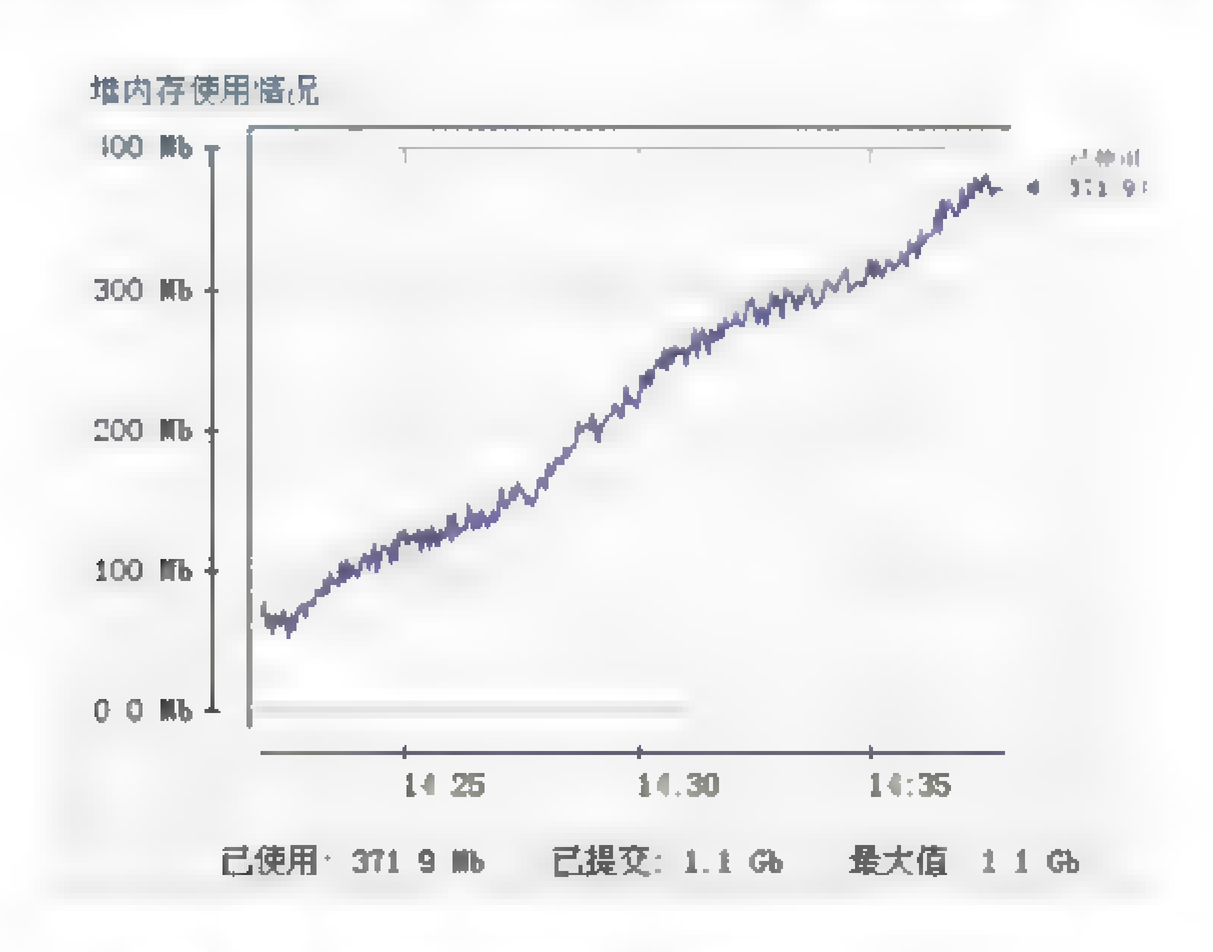


图 10-5 并发收集模式下的堆内存使用情况

在串行收集模式下，运行 10 分钟后的对内存使用情况如图 10-6 所示。
在并行收集模式下，运行 10 分钟后的对内存使用情况如图 10-7 所示。



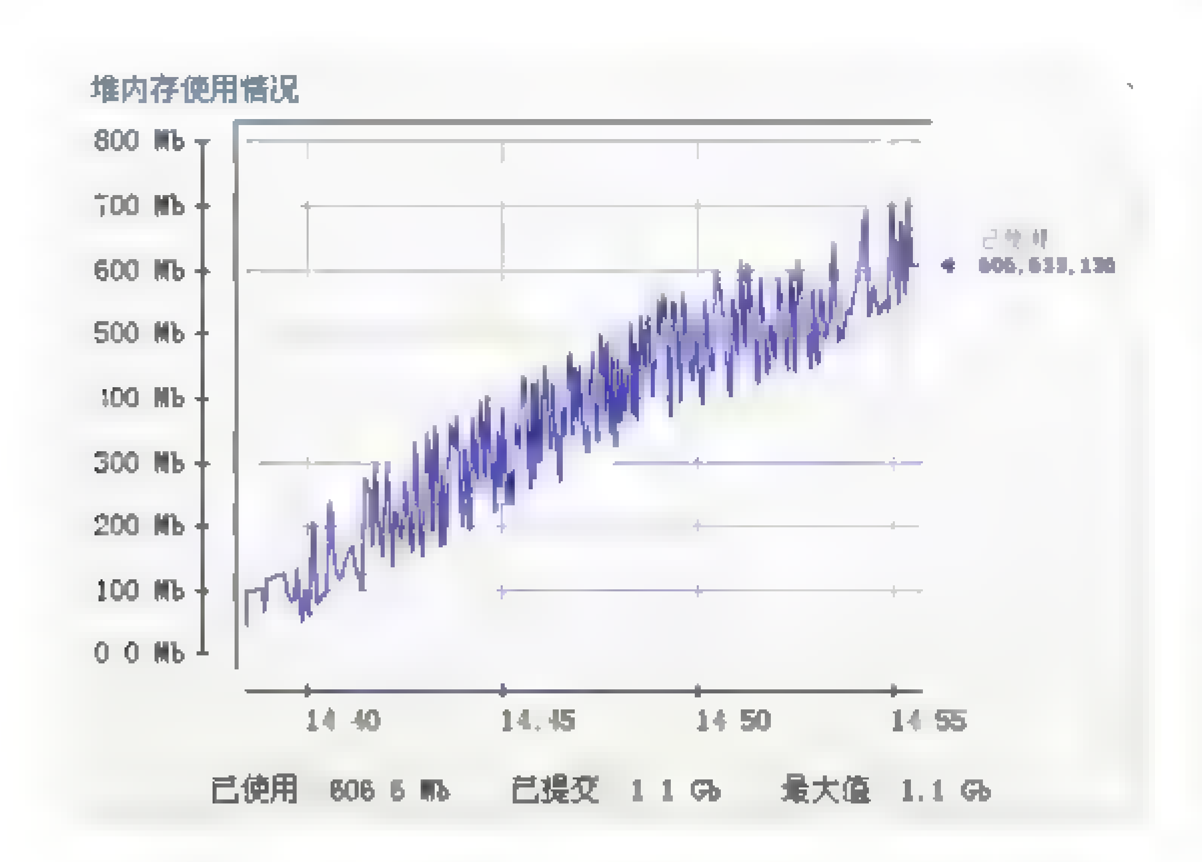


图 10-6　串行收集模式下的堆内存使用情况

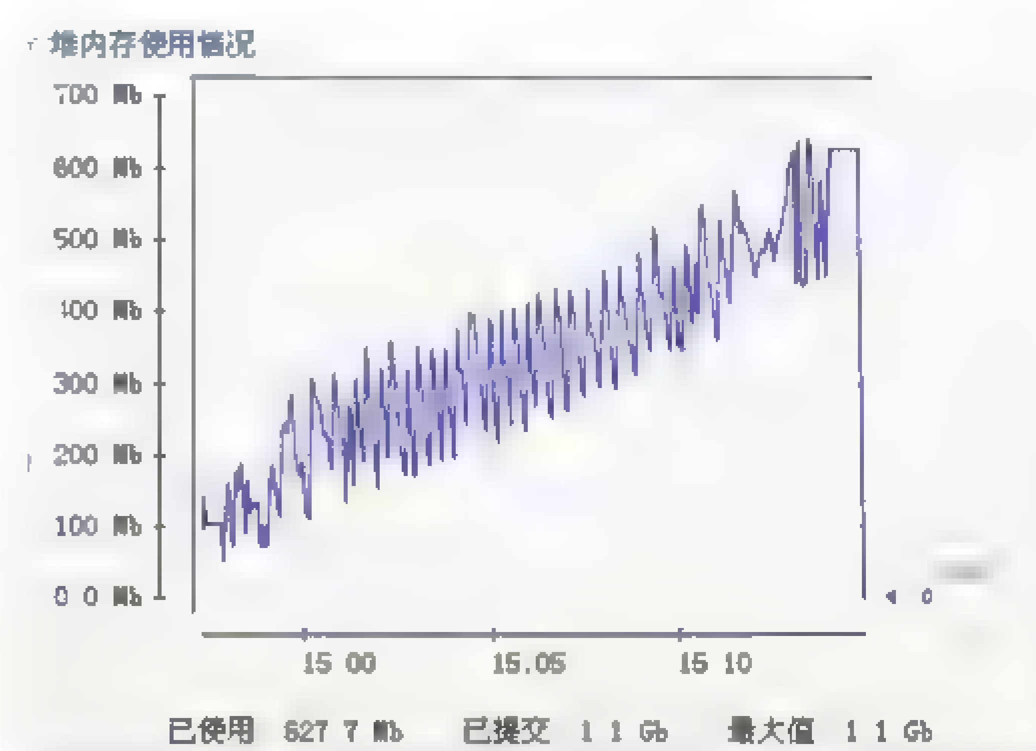


图 10-7　并行收集模式下的堆内存使用情况

30-60：30 个并发用户连续运行 60 分钟的 JVM 内存变化截图如图 10-8 所示。

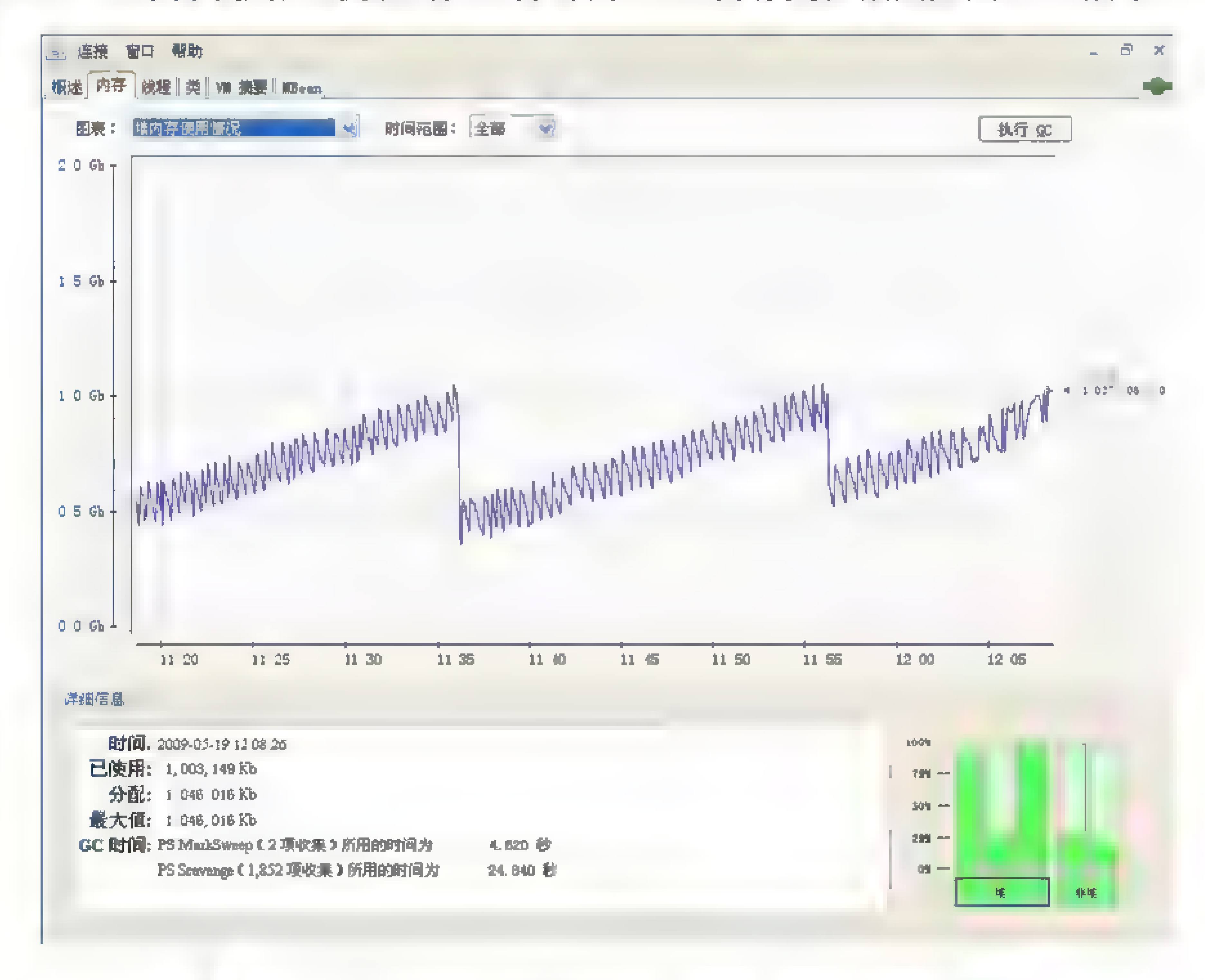


图 10-8　0 个并发用户连续运行 60 分钟的 JVM 内存变化截图

在 11:36 和 11:56 分发生了两次完全 GC(Full GC)，因为这时 PS Old Gen 已经满了，JVM 自动对 Old Gen 中的内存进行了回收。

根据反复的测试并结合被测系统业务特点，最终决定使用以下最优方案进行 8 小时压力测试：

```
JAVA OPTS--server
-Xms1024M
 Xmx1024M
```

```
-Xmn128M
-XX:NewSize=128M
-XX:MaxNewSize=128M
-XX:SurvivorRatio=20
-XX:MaxTenuringThreshold=10
-XX:GCTimeRatio=19
-XX:+UseParNewGC
-XX:+UseConcMarkSweepGC
-XX:+CMSClassUnloadingEnabled
-XX:+UseCMSCompactAtFullCollection
-XX:CMSFullGCsBeforeCompaction=0
-XX:-CMSParallelRemarkEnabled
-XX:CMSInitiatingOccupancyFraction=70
-XX:SoftRefLRUPolicyMSPerMB=0
-XX:PermSize=256m
-XX:MaxPermSize=256m
-Djava.awt.headless=true
```

10.2.6 调优后 JVM 监控图

30Vusers 运行 8 小时后的截图界面分别如图 10-9～图 10-15 所示。

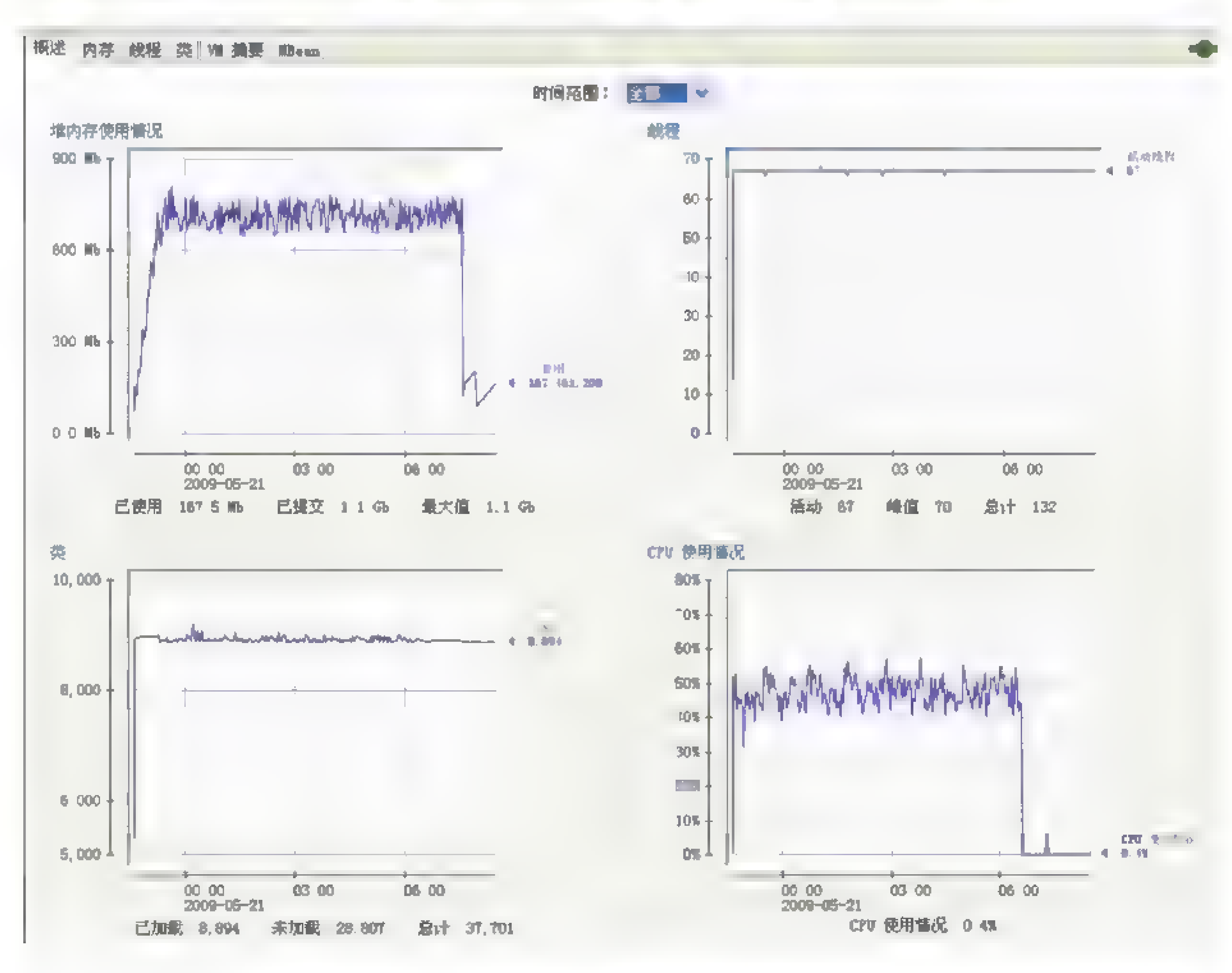


图 10-9 第一个时间段的堆内存使用情况

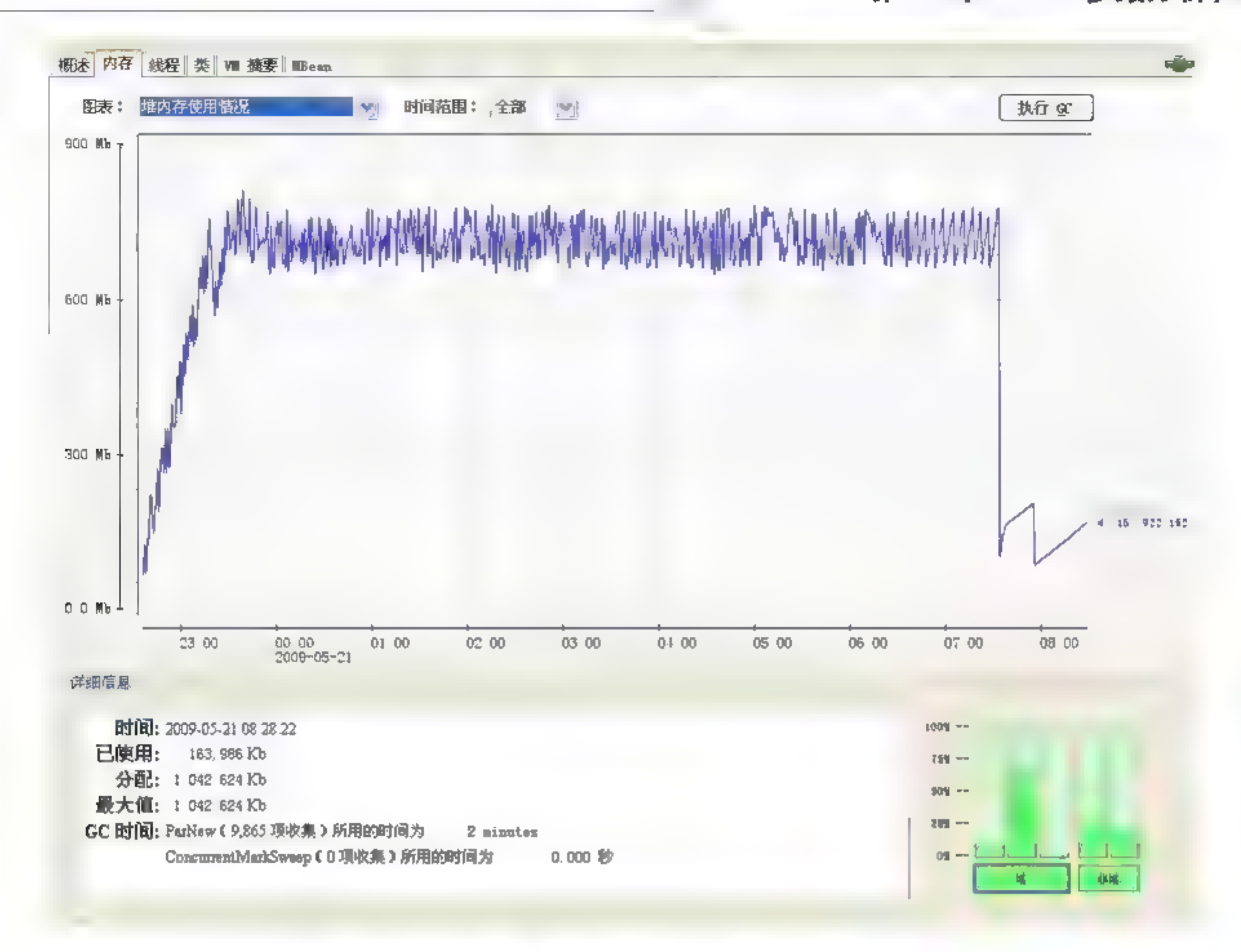


图 10-10　第二个时间段的堆内存使用情况

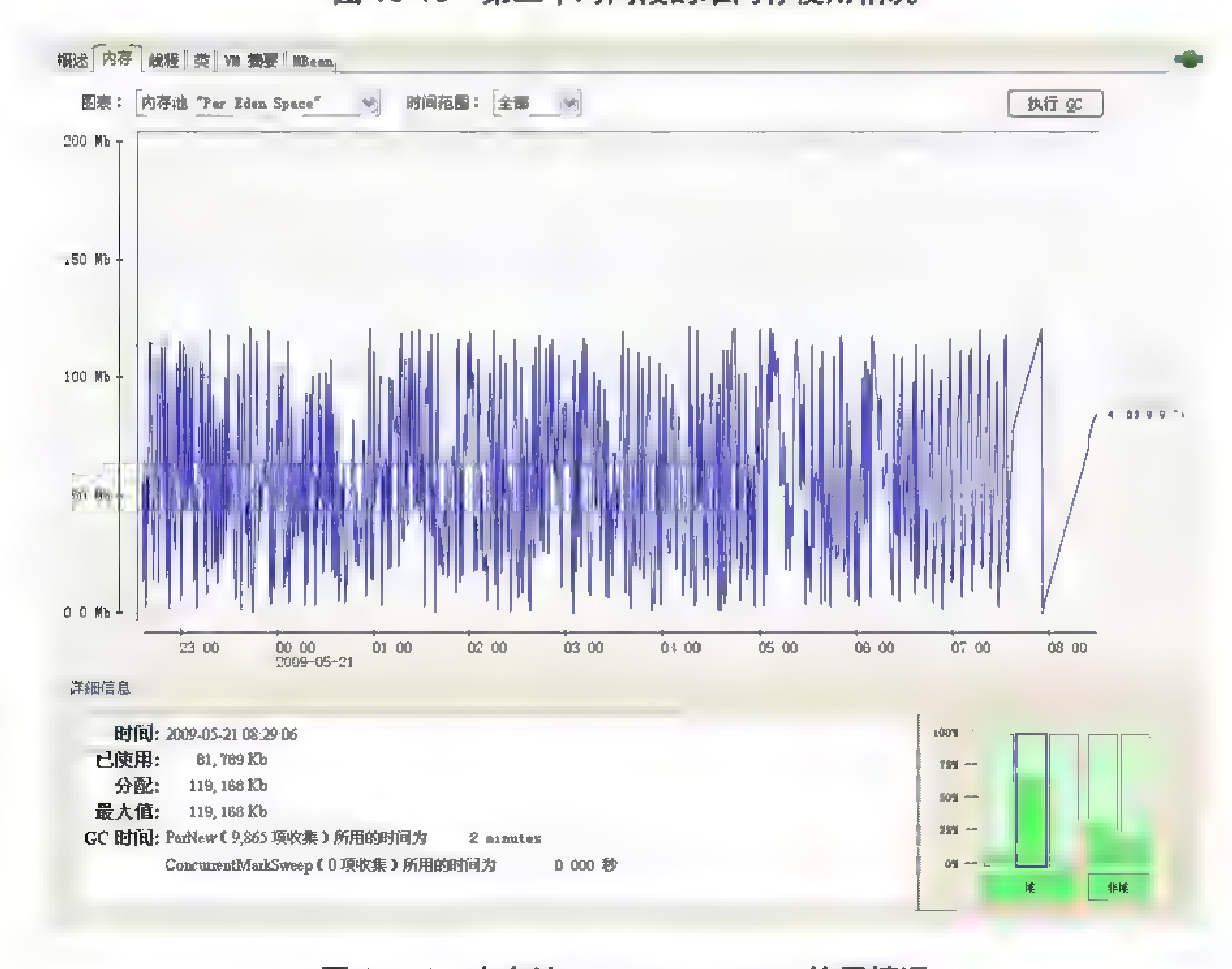


图 10-11　内存池 Par Eden Space 使用情况

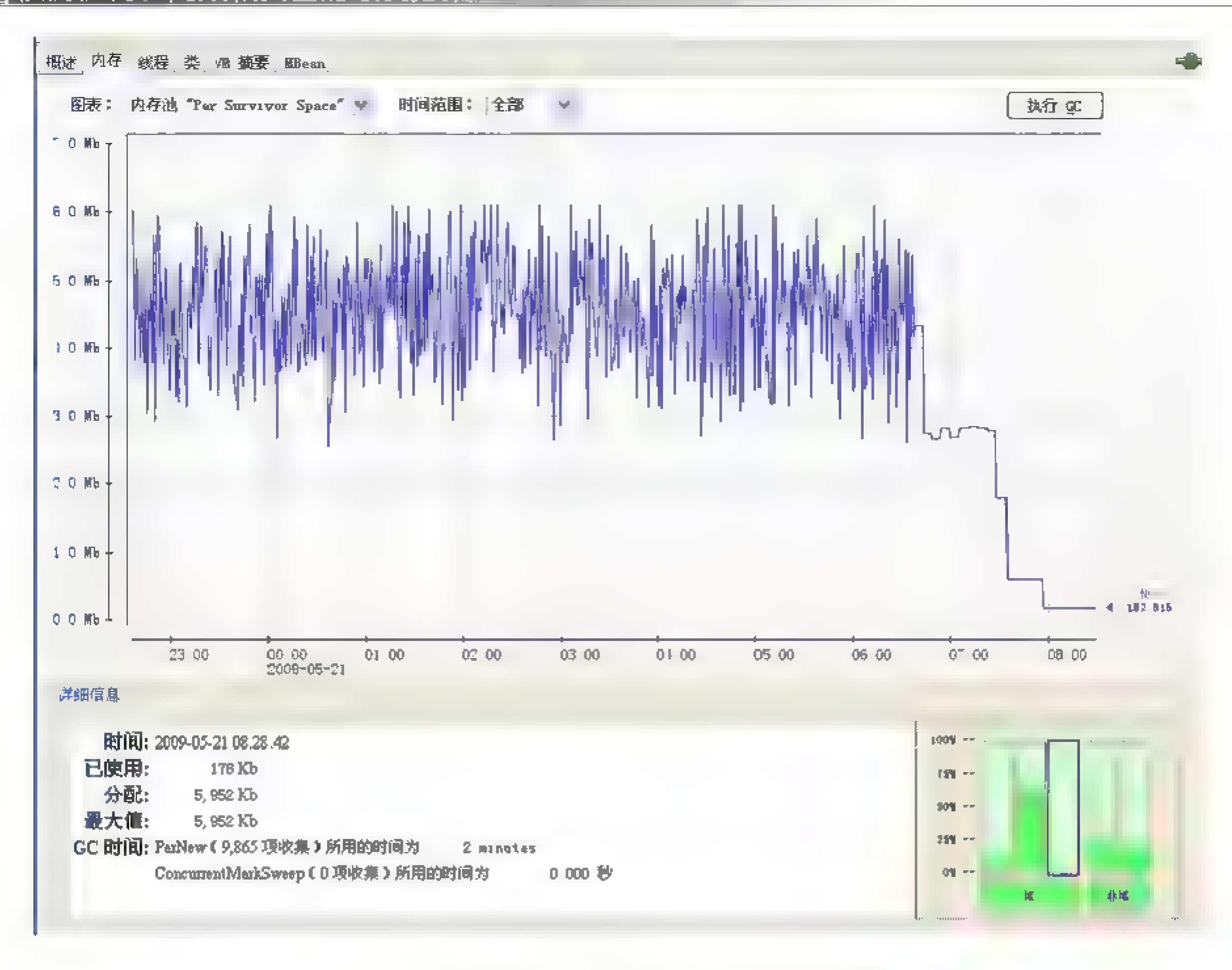


图 10-12　内存池 Par Surviver Space 使用情况

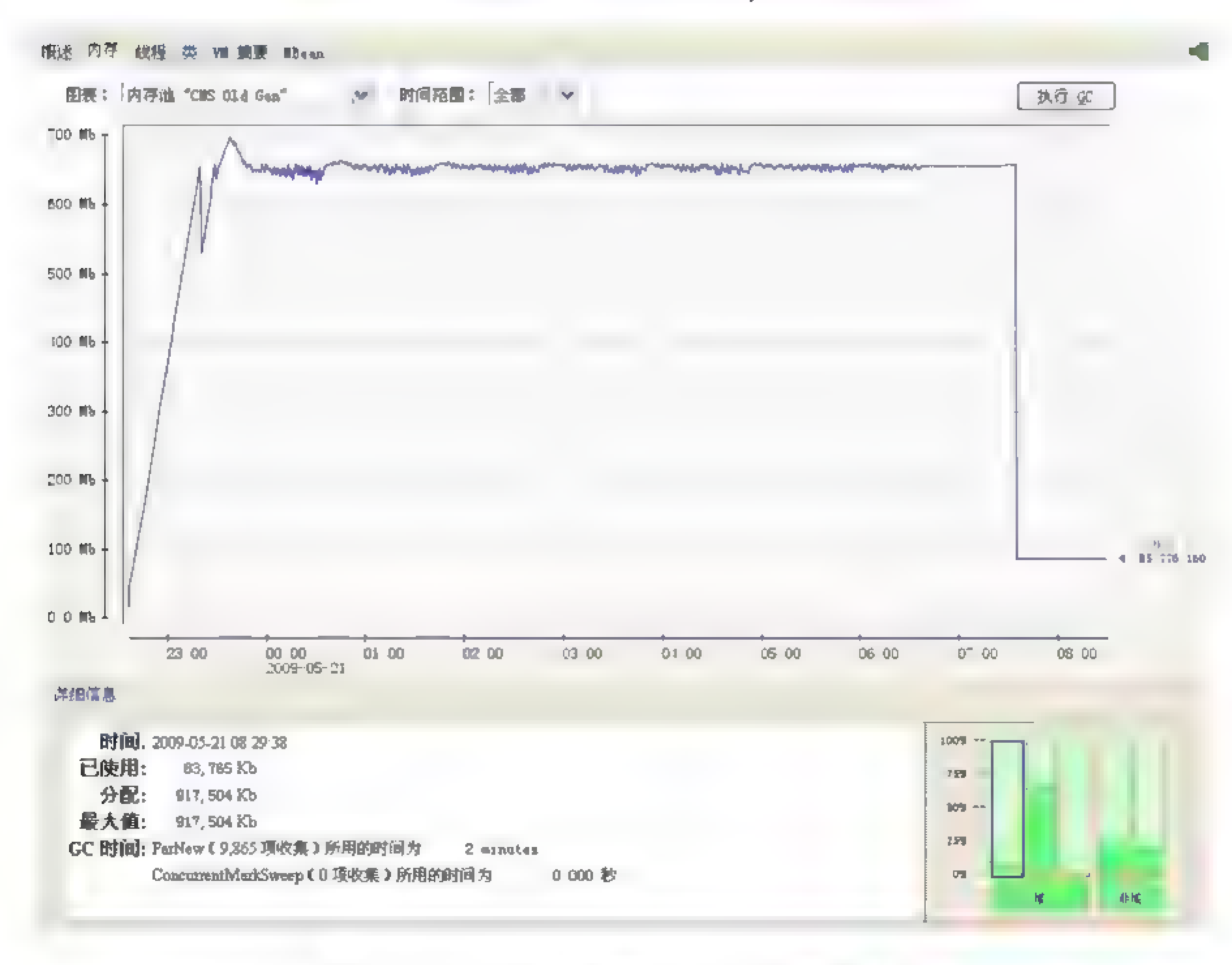


图 10-13　内存池 CMS Old Gen 使用情况

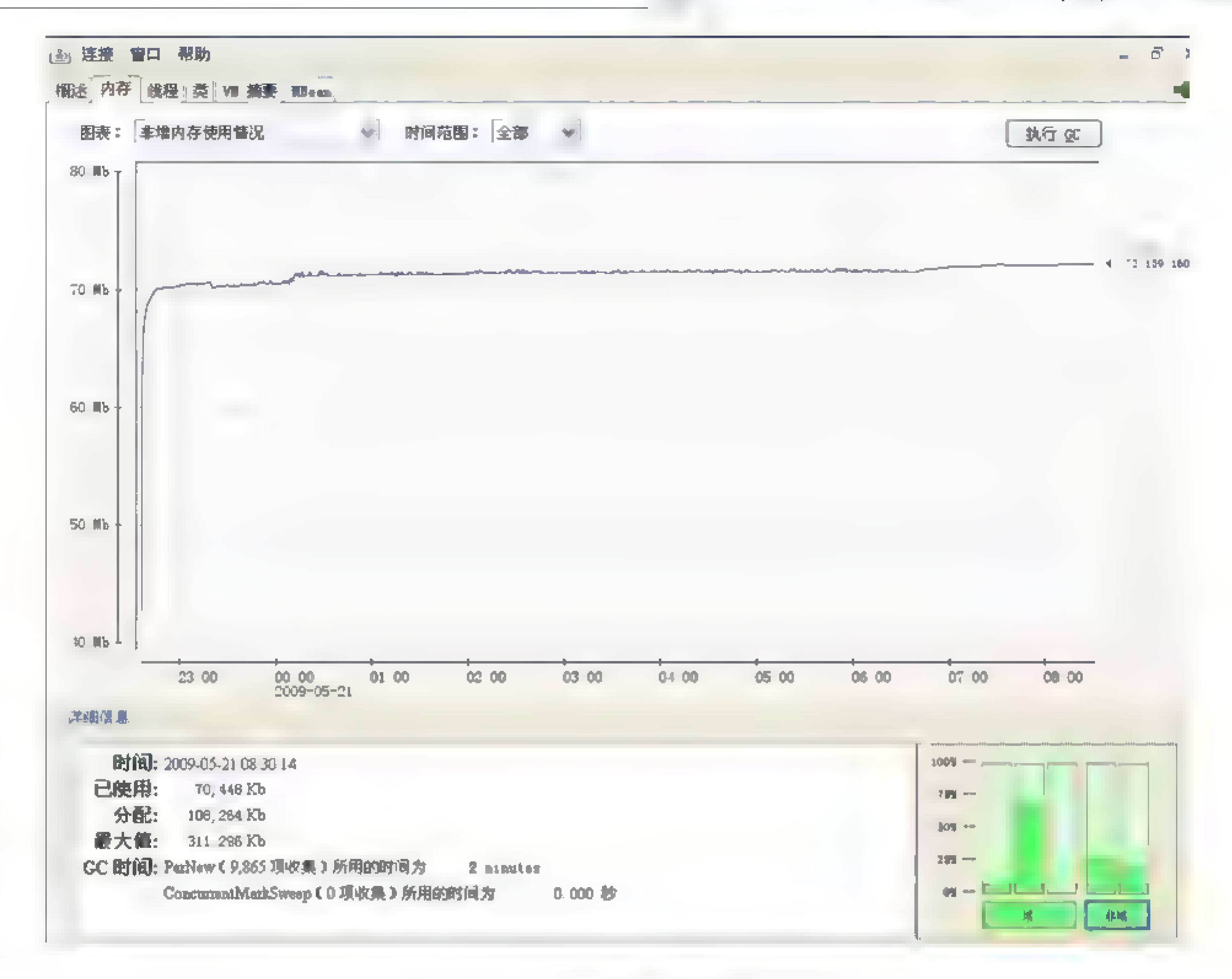


图 10-14　非堆内存使用情况

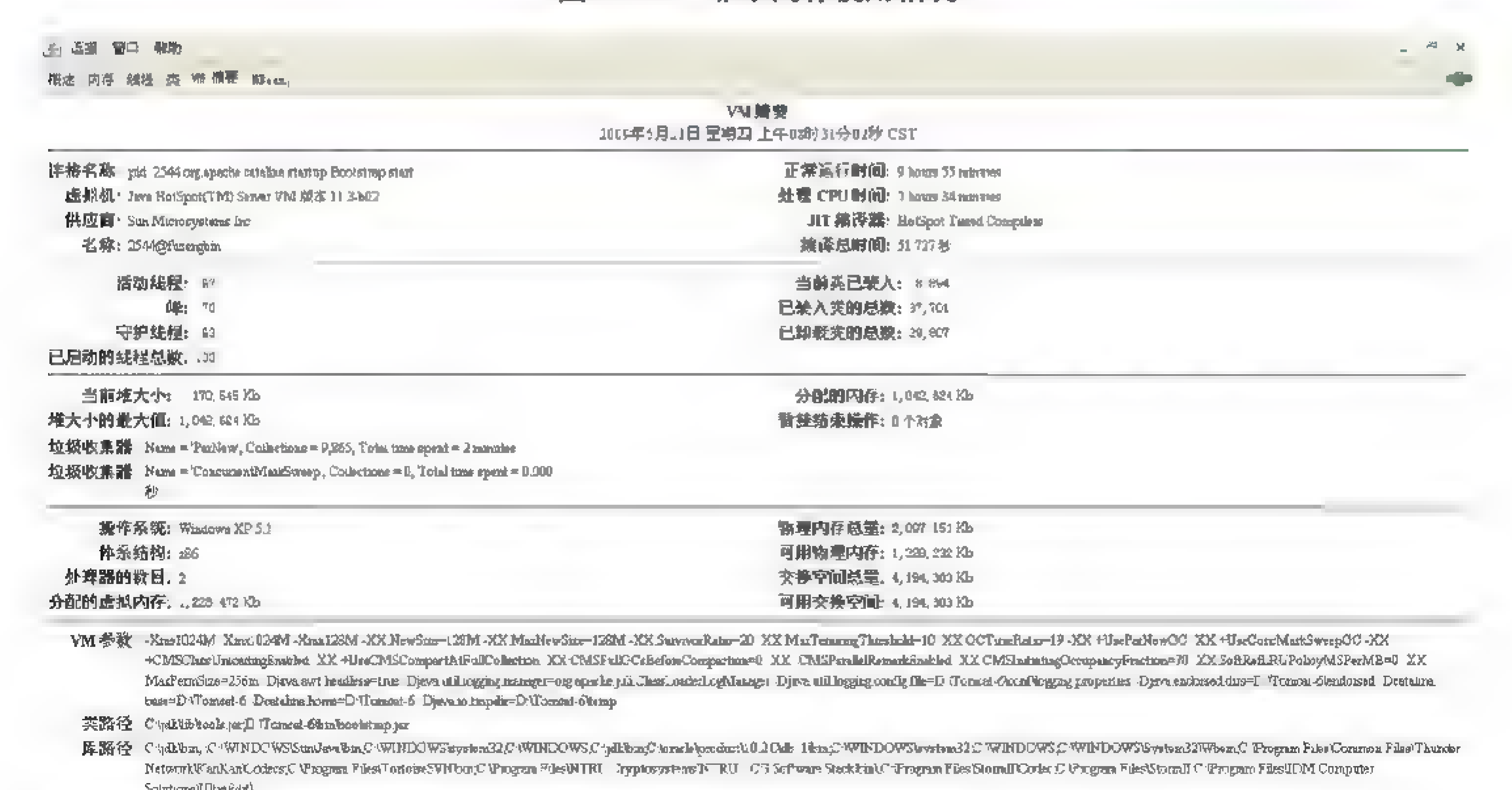


图 10-15　30Vusers 运行 8 小时后的 VM 摘要截图界面

10.2.7 测试结果分析

对于 XX 局采购系统的登录操作来说，当将 JVM 的 NewRatio 和 SurviorRatio 设置成 4 时，性能表现最好。在此基础上在设置-XX:TargetSurvivorRatio 90 和-Xmx1024M 后性能也有一定程度的提升。

10.3 性能问题举例

假设某高校准备正式上线并运行一个系统，每运行一段时间后程序进程会莫名其妙地被 kill 掉，不得不手工启动系统。本节将以此背景为假设，介绍解决性能问题的方案。

10.3.1 查看监控结果

(1) 使用 jmap 命令查看堆内存分配和使用情况：

```
./jmap -heap 31    //31 为程序的进程号
Attaching to process ID 31, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 11.0-b12   //显示 jvm 的版本号
using parallel threads in the new generation.  //说明在年轻代使用了并行收集
using thread-local object allocation.
Concurrent Mark-Sweep GC     //启用 CMS 收集模式

Heap Configuration:
   MinHeapFreeRatio = 40
   MaxHeapFreeRatio = 70   //这两项说明堆内存的使用比例在 30%～60%之间
   MaxHeapSize      = 2147483648 (2048.0MB) //最大堆大小为 2048M
   NewSize          = 805306368 (768.0MB)
   MaxNewSize       = 805306368 (768.0MB)    //年轻代大小为 768M
   OldSize          = 1342177280 (1280.0MB)  //老年代代大小为 1280M
   NewRatio         = 8                      //这个有点自相矛盾，1:8
   SurvivorRatio     = 3                     //救助区大小占整个年轻代的五分之一
   PermSize         = 268435456 (256.0MB)    //持久代大小为 256M
   MaxPermSize      = 268435456 (256.0MB)    //持久代大小为 256M

Heap Usage:
//年轻代大小，这里只计算了一个救助区，所以少了 153M
New Generation (Eden + 1 Survivor Space):
   capacity = 644284416 (614.4375MB)
   used     = 362446760 (345.65616607666016MB)
   free     = 281837656 (268.7813339233984MB)
   56.25570803810968% used
//Eden Space 大小为 614.43-153=460.8M
Eden Space:
   capacity = 483262464 (460.875MB)
   used     = 342975440 (327.0868682861328MB)
   free     = 140287024 (133.7881317138672MB)
   70.97084204743864% used
```

```
//两个救助区的大小均为 153MB， 与前面的 SurvivorRatio 参数设置值计算结果一致。
From Space:
   capacity = 161021952 (153.5625MB)
   used     = 19471320 (18.569297790527344MB)
   free     = 141550632 (134.99320220947266MB)
   12.092338813530219% used
To Space:
   capacity = 161021952 (153.5625MB)
   used     = 0 (0.0MB)
   free     = 161021952 (153.5625MB)
   0.0% used
//老年代大小为 1280M，和根据参数配置计算的结果一致。
concurrent mark-sweep generation:
   capacity = 1342177280 (1280.0MB)
   used     = 763110504 (727.7588882446289MB)
   free     = 579066776 (552.2411117553711MB)
   56.85616314411163% used
//永久代大小为 256M，实际使用不到 50%。可在系统运行一段时间后稳定该值。
Perm Generation:
   capacity = 268435456 (256.0MB)
   used     = 118994736 (113.48222351074219MB)
   free     = 149440720 (142.5177764892578MB)
   44.32899355888367% used
```

(2) 使用 Top 命令监控结果，具体如图 10-16 所示。

```
# top
last pid: 29658;  load avg:  0.46,  0.36,  0.41;       up 240+20:36:06
53 processes: 50 sleeping, 1 zombie, 2 on cpu
CPU states: 85.7% idle, 12.1% user,  2.2% kernel,  0.0% iowait,  0.0% swap
Memory: 8191M phys mem, 3619M free mem, 8001M total swap, 8001M free swap

   PID USERNAME LWP PRI NICE  SIZE   RES STATE    TIME    CPU COMMAND
 29003 root     967  11    0 2601M 2411M cpu/0  487:09 13.26% java
   779 noaccess  28  59    0  183M   99M sleep  134:07  0.02% java
 29658 root       1  59    0 3044K 1736K cpu/1    0:00  0.02% top
   160 root      32  59    0 6324K 3012K sleep   20:32  0.02% nscd
   690 root       1  59    0   25M   12M sleep   52:04  0.01% Xorg
   738 root       1  59    0   11M 7484K sleep   28:48  0.01% dtgreet
 29632 root       1  59    0 8192K 2060K sleep    0:00  0.00% sshd
     7 root      13  59    0   14M   11M sleep    2:14  0.00% svc.startd
   856 root       1  59    0 7184K 1716K sleep    3:39  0.00% sendmail
   352 daemon     2  60  -20 3804K 3032K sleep    5:22  0.00% nfs4cbd
   585 root      17  59    0   13M 8904K sleep   23:39  0.00% fmd
   384 root       4  59    0 4844K 3224K sleep    1:49  0.00% inetd
   393 root       1  59    0 1108K  640K sleep    0:15  0.00% utmpd
   353 daemon    13  59    0  697M  696M sleep    4:05  0.00% nfsmapid
  8485 root       8  59    0 6900K 5664K sleep    1:59  0.00% svc.configd
```

图 10-16　Top 命令监控结果

通过使用 top 命令进行持续监控发现此时 CPU 空闲比例为 85.7%，剩余物理内存为 3619M，虚拟内存 8G 未使用。持续的监控结果显示进程 29003 占用系统内存不断在增加，已经快得到最大值。

(3) 使用 Jstat 命令监控结果，具体如图 10-17 所示。

```
# ./jstat -gcutil 29003 1000 100
  S0     S1     E      O      P     YGC     YGCT    FGC    FGCT     GCT
  0.00   0.00  41.49  80.49  48.41   1635   35.472 21928 7327.744 7363.216
  0.00   0.00  41.51  80.49  48.41   1635   35.472 21929 7328.064 7363.536
  0.00   0.00  41.57  80.49  48.41   1635   35.472 21929 7328.064 7363.536
  0.00   0.00  41.57  80.49  48.41   1635   35.472 21930 7328.282 7363.754
  0.00   0.00  41.58  80.49  48.41   1635   35.472 21930 7328.282 7363.754
  0.00   0.00  41.59  80.49  48.41   1635   35.472 21930 7328.282 7363.754
  0.00   0.00  41.59  80.49  48.41   1635   35.472 21930 7328.282 7363.754
  0.00   0.00  41.61  80.49  48.41   1635   35.472 21930 7328.282 7363.754
  0.00   0.00  41.61  80.49  48.41   1635   35.472 21930 7328.282 7363.754
  0.00   0.00  41.62  80.49  48.41   1635   35.472 21931 7328.282 7363.754
  0.00   0.00  41.62  80.48  48.41   1635   35.472 21932 7328.818 7364.290
  0.00   0.00  41.69  80.48  48.41   1635   35.472 21932 7328.818 7364.290
  0.00   0.00  41.71  80.48  48.41   1635   35.472 21932 7328.818 7364.290
  0.00   0.00  41.73  80.48  48.41   1635   35.472 21932 7328.818 7364.290
  0.00   0.00  41.73  80.48  48.41   1635   35.472 21932 7328.818 7364.290
  0.00   0.00  41.81  80.48  48.41   1635   35.472 21932 7328.818 7364.290
  0.00   0.00  41.84  80.48  48.41   1635   35.472 21933 7329.139 7364.611
  0.00   0.00  41.91  80.48  48.41   1635   35.472 21933 7329.139 7364.611
```

图 10-17　使用 Jstat 命令监控结果

使用 jstat 命令对 PID 为 29003 的进程进行 gc 回收情况检查，发现由于 Old 段的内存使用量已经超过了设定的 80%的警戒线，导致系统每隔一两秒就进行一次 FGC，FGC 的次数明显多余 YGC 的次数，但是每次 FGC 后 old 的内存占用比例却没有明显变化—系统尝试进行 FGC 也不能有效地回收这部分对象所占内存。同时也说明年轻代的参数配置可能有问题，导致大部分对象都不得不放到老年代来进行 FGC 操作，这个或许跟系统配置的会话失效时间过长有关。

(4) 使用 Jstack 打印出的堆栈内容，具体如图 10-18 所示。

```
"EventMonitorThread[com.zzxy.willow.core.StandardContextDeployer@1bda9b9 - ZJGXSQ]" daemon prio=3 tid=0x
   java.lang.Thread.State: WAITING (on object monitor)
  at java.lang.Object.wait(Native Method)
  - waiting on <0xcc0340d8> (a com.zzxy.workflow.impl.monitor.EventQueue)
  at java.lang.Object.wait(Object.java:485)
  at com.zzxy.workflow.impl.monitor.EventQueue.getNextEvent(EventQueue.java:73)
  - locked <0xcc0340d8> (a com.zzxy.workflow.impl.monitor.EventQueue)
  at com.zzxy.workflow.impl.monitor.EventMonitorThread.run(EventMonitorThread.java:45)

"EventMonitorThread[com.zzxy.willow.core.StandardContextDeployer@1bda9b9 - 评审专家信息维护审批流程]" da
   java.lang.Thread.State: WAITING (on object monitor)
  at java.lang.Object.wait(Native Method)
  - waiting on <0xcc0365d8> (a com.zzxy.workflow.impl.monitor.EventQueue)
  at java.lang.Object.wait(Object.java:485)
  at com.zzxy.workflow.impl.monitor.EventQueue.getNextEvent(EventQueue.java:73)
  - locked <0xcc0365d8> (a com.zzxy.workflow.impl.monitor.EventQueue)
  at com.zzxy.workflow.impl.monitor.EventMonitorThread.run(EventMonitorThread.java:45)

"EventMonitorThread[com.zzxy.willow.core.StandardContextDeployer@1bda9b9 - 采购计划审批]" daemon prio=3
   java.lang.Thread.State: WAITING (on object monitor)
  at java.lang.Object.wait(Native Method)
  - waiting on <0xcc036628> (a com.zzxy.workflow.impl.monitor.EventQueue)
  at java.lang.Object.wait(Object.java:485)
  at com.zzxy.workflow.impl.monitor.EventQueue.getNextEvent(EventQueue.java:73)
  - locked <0xcc036628> (a com.zzxy.workflow.impl.monitor.EventQueue)
```

图 10-18　使用 Jstack 打印出的堆栈内容

在图 10-18 中会发现大量的的工作流线程锁定，在图 10-19 中也会发现大量的 CMS 线程池管理线程锁定。

```
"Thread-52390" daemon prio=3 tid=0x0bbdf400 nid=0xd60a in Object.wait()
   java.lang.Thread.State: TIMED_WAITING (on object monitor)
  at java.lang.Object.wait(Native Method)
  - waiting on <0xddd59d38> (a com.web.cms.infopub.publish.ThreadPool)
  at com.web.cms.infopub.publish.ThreadPool.run(ThreadPool.java:140)
  - locked <0xddd59d38> (a com.web.cms.infopub.publish.ThreadPool)
  at java.lang.Thread.run(Thread.java:619)

"Thread-52261" daemon prio=3 tid=0x0a423000 nid=0xd582 in Object.wait()
   java.lang.Thread.State: TIMED_WAITING (on object monitor)
  at java.lang.Object.wait(Native Method)
  - waiting on <0xddb91780> (a com.web.cms.infopub.publish.ThreadPool)
  at com.web.cms.infopub.publish.ThreadPool.run(ThreadPool.java:140)
  - locked <0xddb91780> (a com.web.cms.infopub.publish.ThreadPool)
  at java.lang.Thread.run(Thread.java:619)

"Thread-52257" daemon prio=3 tid=0x0adf2000 nid=0xd57e in Object.wait()
   java.lang.Thread.State: TIMED_WAITING (on object monitor)
  at java.lang.Object.wait(Native Method)
  - waiting on <0xddb90200> (a com.web.cms.infopub.publish.ThreadPool)
  at com.web.cms.infopub.publish.ThreadPool.run(ThreadPool.java:140)
  - locked <0xddb90200> (a com.web.cms.infopub.publish.ThreadPool)
  at java.lang.Thread.run(Thread.java:619)
```

图 10-19　大量的 CMS 线程池管理线程锁定

10.3.2　原因分析

通过对 JVM 内存进行实时监控后发现导致老年代内存不能有效回收的原因就在于堆栈中存在大量的线程死锁问题。建议开发组认真审查源代码，看看是否存在线程死锁的缺陷。

假设该系统的 JVM 设置如下：

```
<jvm-options>-XX:+PrintGCApplicationConcurrentTime</jvm-options> <jvm-options>-XX:+PrintGCApplicationStoppedTime</jvm-options>
<jvm-options>-XX:+PrintGCTimeStamps</jvm-options>
<jvm-options>-XX:+PrintGCDetails</jvm-options>
<jvm-options>-Xms2048m</jvm-options>
<jvm-options>-Xmx2048m</jvm-options>
<jvm-options>-server</jvm-options>
<jvm-options>-Djava.awt.headless=true</jvm-options>
<jvm-options>-XX:PermSize=256m</jvm-options>
<jvm-options>-XX:MaxPermSize=256m</jvm-options>
<jvm-options>-XX:+DisableExplicitGC</jvm-options>
<jvm-options>-Xmn768M</jvm-options>
<jvm-options>-XX:SurvivorRatio=3</jvm-options>
<jvm-options>-Xss128K</jvm-options>
<jvm-options>-XX:TargetSurvivorRatio=80</jvm-options>
<jvm-options>-XX:MaxTenuringThreshold=5</jvm-options>
<jvm-options>-XX:+UseConcMarkSweepGC</jvm-options>
<jvm-options>-XX:+CMSClassUnloadingEnabled</jvm-options>
<jvm-options>-XX:+UseCMSCompactAtFullCollection</jvm-options>
<jvm-options>-XX:-CMSParallelRemarkEnabled</jvm-options>
```

由此可见，性能调优要做到有的放矢，根据实际业务系统的特点，以一定时间的 JVM 日志记录为依据，进行有针对性的调整、比较和观察。性能调优是个无止境的过程，要综合权衡调优成本和更换硬件成本的大小，使用最经济的手段达到最好的效果。性能调优不

仅仅包括 JVM 的调优，还有服务器硬件配置、操作系统参数、中间件线程池、数据库连接池、数据库本身参数以及具体的数据库表、索引、分区等的调整和优化。通过特定工具检查代码中存在的性能问题，并加以修正是一种比较经济快捷的调优方法。

10.4 调优案例分析

本章中的案例大部分来源于笔者处理过的一些问题，还有一小部分来源于网上有特色和代表性的案例总结。出于对客户商业信息保护的目的，在不影响前后逻辑的前提下，笔者对实际环境和用户业务做了一些屏蔽和精简。

10.4.1 高性能硬件上的程序部署策略

一个 15 万 PV/天左右的在线文档类型网站最近更换了硬件系统，新的硬件为 4 个 CPU、16GB 物理内存，操作系统为 64 位 CentOS 5.4，Resin 作为 Web 服务器。整个服务器暂时没有部署别的应用，所有硬件资源都可以提供给访问量并不算太大的网站使用。管理员为了尽量利用硬件资源选用了 64 位的 JDK 1.5，并通过-Xmx 和-Xms 参数将 Java 堆固定在 12GB。使用一段时间后发现使用效果并不理想，网站经常不定期出现长时间没有响应的现象。

监控服务器运行状况后发现网站没有响应是由 GC 停顿导致的，虚拟机运行在 Server 模式，默认使用吞吐量优先收集器，回收 12GB 的堆，一次 Full GC 的停顿时间高达 14 秒。并且由于程序设计的关系，访问文档时要把文档从磁盘提取到内存中，导致内存中出现很多由文档序列化产生的大对象，这些大对象很多都进入了老年代，没有在 Minor GC 中清理掉。这种情况下即使有 12GB 的堆，内存也很快会被消耗殆尽，由此导致每隔十几分钟出现十几秒的停顿，令网站开发人员和管理员感到很沮丧。

这里先不延伸讨论程序代码问题，程序部署上的主要问题显然是过大的堆内存进行回收时带来的长时间的停顿。硬件升级前使用 32 位系统 1.5GB 的堆，用户只感到访问网站比较缓慢，但不会发生十分明显的停顿，因此才考虑升级硬件提升程序效能，如果重新缩小给 Java 堆分配的内存，那么硬件上的投资就浪费了。

在高性能硬件上部署程序，目前主要有两种方式：

- 通过 64 位 JDK 来使用大内存。
- 使用若干个 32 位虚拟机建立逻辑集群来利用硬件资源。

此案例中的管理员采用了第一种部署方式。对于用户交互性强、对停顿时间敏感的系统，可以给 Java 虚拟机分配超大堆的前提是有把握把应用程序的 Full GC 频率控制得足够低，至少要低到不会影响用户使用，譬如十几个小时乃至一天才出现一次 Full GC，这样可以通过在深夜执行定时任务的方式触发 Full GC 甚至自动重启应用服务器来将内存可用空间保持在一个稳定的水平。

控制 Full GC 频率的关键是看应用中绝大多数对象能否符合“朝生夕灭”的原则，即大多数对象的生存时间不应当太长，尤其是不能产生成批量的、长生存时间的大对象，这

样才能保障老年代空间的稳定。

在大多数网站形式的应用里，主要对象的生存周期都应该是请求级或页面级的，会话级和全局级的长生命对象相对很少。只要代码写得合理，应当都能实现在超大堆中正常使用而没有 Full GC，这样的话，使用超大堆内存时，网站响应的速度才比较有保证。除此之外，如果读者计划使用 64 位 JDK 来管理大内存，还需要考虑下面可能面临的问题：

- 内存回收导致的长时间停顿。
- 现阶段，64 位 JDK 的性能测试结果普遍低于 32 位 JDK。
- 需要保证程序足够稳定，因为这种应用要是产生堆溢出几乎就无法产生堆转储快照(因为要产生十几 GB 乃至更大的 dump 文件)，哪怕产生了快照也几乎无法进行分析。
- 相同的程序在 64 位 JDK 中消耗的内存一般比 32 位 JDK 大，这是由指针膨胀及数据类型对齐补白等因素导致的。

上面的问题听起来有点吓人，所以现阶段不少管理员还是选择第二种方式：使用若干个 32 位虚拟机建立逻辑集群来利用硬件资源。具体做法是在一台物理机器上启动多个应用服务器进程，给每个服务器进程分配不同的端口，然后在前端搭建一个负载均衡器，以反向代理的方式来分配访问请求。读者不需要太在意均衡器转发所消耗的性能，即使使用 64 位 JDK，许多应用也不止有一台服务器，因此在许多应用中前端的均衡器总是要存在的。

考虑到在一台物理机器上建立逻辑集群的目的仅仅是尽可能地利用硬件资源，并不需要关心状态保留、热转移之类的高可用性需求，也不需要保证每个虚拟机进程有绝对准确的均衡负载，因此使用无 Session 复制的亲合式集群是一个相当不错的选择。我们仅仅需要保障集群具备亲和性，也就是均衡器按一定的规则算法(一般根据 SessionID 分配)将一个固定的用户请求永远分配到固定的一个集群节点进行处理即可，这样程序开发阶段就基本不用为集群环境做什么特别的考虑。

当然，很少有没有缺点的方案，如果读者计划使用逻辑集群的方式来部署程序，可能会遇到下面一些问题：

- 尽量避免节点竞争全局的资源，最典型的就是磁盘竞争，各个节点如果同时访问某个磁盘文件的话(尤其是并发写操作容易出现问题)，很容易导致 IO 异常。
- 很难最高效率地利用某些资源池，譬如连接池，一般都是在各个节点建立自己独立的连接池，这样有可能导致一些节点池满了而另外一些节点仍有较多空余。尽管可以使用集中式的 JNDI，但这有一定的复杂性并且可能带来额外的性能代价。
- 各个节点仍然不可避免地受到 32 位的内存限制，在 32 位 Windows 平台中每个进程只能使用 2GB 的内存，考虑到堆以外的内存开销，堆一般最多只能开到 1.5GB。在某些 Linux、UNIX 系统(如 Solaris)中，可以提升到 3GB 乃至接近 4GB 的内存，但 32 位中仍然受最高 4GB (232)内存的限制。

大量使用本地缓存(如大量使用 HashMap 作为 KN 缓存)的应用，在逻辑集群中会造成较大的内存浪费，因为每个逻辑节点上都有一份缓存，这时可以考虑把本地缓存改为集中

式缓存。

介绍完这两种部署方式，再重新回到这个案例之中，最后的部署方案调整为建立 5 个 32 位 JDK 的逻辑集群，每个进程按 2GB 内存计算(其中堆固定为 1.5GB)，占用了 10GB 的内存。另外建立一个 Apache 服务作为前端均衡代理访问门户。考虑到用户对响应速度比较关心，并且文档服务的主要压力集中在磁盘和内存访问上，CPU 资源敏感度较低，因此改为 CMS 收集器进行垃圾回收。部署方式调整后，服务再没有出现长时间停顿，速度比硬件升级前有较大提升。

10.4.2 堆外内存导致的溢出错误

这是一个学校的小型项目：基于 B/S 的电子考试系统，为了实现客户端能实时地从服务端接收考试数据，系统使用了逆向 AJAX 技术(也称为 Comet 或 Server Side Push)，选用 CometD 1.1.1 作为服务端推送框架，服务器是 Jetty 7.1.4，硬件为一台普通 PC(Core i5 CPU，4GB 内存，运行 32 位 Windows 操作系统)。

测试期间发现服务端不定时抛出内存溢出异常，服务器不一定每次都会出现异常，但假如正式考试时崩溃一次，那估计整场电子考试都会乱套，网站管理员尝试过把堆开到最大，32 位系统最多到 1.6GB 基本无法再加大了，而且开大了也基本没效果，抛出内存溢出异常好像更加频繁了。加入-XX:+HeapDumpOnOutOfMemoryError，居然也没有任何反应，抛出内存溢出异常时什么文件都没有产生。无奈之下只好挂着 jstat 使劲盯屏幕，发现 GC 并不频繁，Eden 区、Survivor 区、老年代及永久代内存全部都表示“情绪稳定，压力不大”，但照样不停地抛出内存溢出异常，管理员压力很大。最后，在内存溢出后从系统日志中找到异常堆栈，代码清单如下。

```
Lorg.eclipse.jetty.util.log] handle failed java.lang.OutOfMemoryError: null
at  sun.misc.Unsafe.allocateMemory (Native Method)
at java.nio.DirectByteBuffer. <init>( DirectByteBuffer.java:99)
at java.nio.ByteBuffer.allocateDirect(ByteBuffer.java:288)
at  org.eclipse.jetty.io.nio.DirectNIOBuffer.<init>
```

我们知道操作系统对每个进程能管理的内存是有限制的，这台服务器使用的 32 位 Windows 平台的限制是 2GB，其中给了 Java 堆 1.6GB，而 Direct Memory 并不算在 1.6GB 的堆之内，因此它只能在剩余的 0.4GB 空间中分出一部分。在此应用中导致溢出的关键是：垃圾收集进行时，虚拟机虽然会对 Direct Memory 进行回收，但是 Direct Memory 却不能像新生代和老年代那样，发现空间不足了就通知收集器进行垃圾回收，它只能等待老年代满了后 Full GC，然后“顺便地”帮它清理掉内存的废弃对象。否则，它只能等到抛出内存溢出异常时，先 catch 掉，再在 catch 块里面“大喊”一声：“System.gc()!”。要是虚拟机还是不听(譬如打开了-XX:+DisableExplicitGC 开关)，那就只能眼睁睁地看着堆中还有许多空闲内存，自己却不得不抛出内存溢出异常了。而本案例中使用的 CometD 1.1.1 框架，正好有大量的 NIO 操作需要用到 Direct Memory。

从实践经验的角度出发，除了 Java 堆和永久代之外，我们注意到下面这些区域还会占用较多的内存，这里所有的内存总和会受到操作系统进程最大内存的限制：

- Direct Memory：可通过-XX:MaxDirectMemorySize 调整大小，内存不足时抛出

OutOfMemoryError 或 OutOfMemoryError: Direct buffer memory。

- 线程堆栈：可通过-Xss 调整大小，内存不足时抛出 StackOverflowError(纵向无法分配，即无法分配新的栈帧)或 OutOfMemoryError: unable to create new native thread(横向无法分配，即无法建立新的线程)。
- Socket 缓存区：每个 Socket 连接都 Receive 和 Send 两个缓存区，分别占大约 37KB 和 25KB 的内存，连接多的话这块内存占用也比较可观。如果无法分配，则可能会抛出 IOException:Too many open files 异常。
- JNI 代码：如果代码中使用 JNI 调用本地库，那本地库使用的内存也不在堆中。
- 虚拟机和 GC：虚拟机和 GC 的代码执行也要消耗一定的内存。

10.4.3 外部命令导致系统缓慢

这是一个来自网络的案例：一个数字校园应用系统，运行在一台 4 个 CPU 的 Solaris 10 操作系统上，中间件为 GlassFish 服务器。系统在进行大并发压力测试的时候，发现请求响应时间比较慢，通过操作系统的 mpstat 工具发现 CPU 使用率很高，并且占用绝大多数 CPU 资源的程序并不是应用系统本身。这是个不正常的现象，通常情况下用户应用的 CPU 占用率应该占主要地位，才能说明系统是正常工作的。

通过 Solaris 10 的 Dtrace 脚本可以查看当前情况下哪些系统调用花费了最多的 CPU 资源，Dtrace 运行后发现最消耗 CPU 资源的竟然是“fork”系统调用。众所周知，fork 系统调用是 Linux 用来产生新进程的，在 Java 虚拟机中，用户编写的 Java 代码最多只有线程的概念，不应当有进程的产生。

这是个非常异常的现象。通过本系统的开发人员最终找到了答案：每个用户请求的处理都需要执行一个外部 shell 脚本来获得系统的一些信息。执行这个 shell 脚本是通过 Java 的 Runtime.getRuntime().exec()方法来调用的。这种调用方式可以达到目的，但是它在 Java 虚拟机中非常消耗资源，即使外部命令本身能很快执行完毕，频繁调用时创建进程的开销也非常可观。Java 虚拟机执行这个命令的过程是：首先克隆一个和当前虚拟机拥有一样环境变量的进程，再用这个新的进程去执行外部命令，最后再退出这个进程。如果频繁执行这个操作，系统的消耗会很大，不仅是 CPU，内存的负担也很重。

用户根据建议去掉这个 shell 脚本执行的语句，改为使用 Java 的 API 去获取这些信息后，系统很快就恢复了正常。

10.4.4 服务器 JVM 进程崩溃

一个基于 B/S 的 MIS 系统，硬件为两台 2 个 CPU、8GB 内存的 HP 系统，服务器是 WebLogic 9.2(就是第二个案例中的那套系统)。正常运行一段时间后，最近发现在运行期间频繁出现集群节点的虚拟机进程自动关闭的现象，留下了一个 hs_err_pid###.log 文件后，进程就消失了，两台物理机里的每个节点都出现过进程崩溃的现象。从系统日志中注意到，每个节点的虚拟机进程在崩溃前不久，都发生过大量相同的异常。

代码清单 10-1　异常堆栈

```
java.net.SocketException: Connection reset
fat j ava.net.SocketInputStream. read(SocketInputStream.java:168)
}at j ava .io .Buf feredInputStream.fill( BufferedInputStream.java:218)
   atj ava .io. Buf feredInputStream.read (BufferedInputStream.java:235)
   at   org.
apache.axis.transport .http.HTTPSender.readHeadersFromSocket(HTTPSender.java:583)
   at  org. apache.axis.transport.http.HTTPSender.invoke( HTTPSender.java:143)
   …99 more
```

这是一个远端断开连接的异常，通过系统管理员了解到系统最近与一个 OA 门户做了集成，在 MIS 系统工作流的待办事项变化时，要通过 Web 服务通知 OA 门户系统，把待办事项的变化同步到 OA 门户之中。通过 SoapUI 测试了一下同步待办事项的几个 Web 服务，发现调用后竟然需要长达 3 分钟才能返回，并且返回的结果都是连接中断。

由于 MIS 系统的用户多，待办事项变化很快，为了不被 OA 系统的速度拖累，使用了异步的方式调用 Web 服务，但由于两边服务的速度完全不对等，时间越长就累积了越多 Web 服务没有调用完成，导致在等待的线程和 Socket 连接越来越多，最终超过虚拟机的承受能力后使得虚拟机进程崩溃。通知 OA 门户方修复无法使用的集成接口，并将异步调用改为生产者/消费者模式的消息队列实现后，系统恢复正常。

10.5　Eclipse 调优

很多 Java 开发人员都有这样一种观念：系统调优的工作都是针对服务端应用而言的，规模越大的系统，需要越专业的调优运维团队参与。这个观点不能说不对，上一节中笔者所列举的案例确实都是服务端运维和调优的例子，但服务端应用需要调优，并不说明其他应用就不需要了，作为一个普通的 Java 开发人员，前面讲的各种虚拟机的原理和最佳实践的方法距离我们并不遥远，开发者身边的很多场景都可以使用上面这些知识。下面就通过一个普通程序员日常工作中可以随时接触到的开发工具开始这次实战。

10.5.1　Eclipse 快捷键

- Ctrl+Shift+L：显示 Eclipse 的快捷键。
- ctrl+shift+o：自动引入所依赖的类。
- Ctrl+1：快速修复，这是最经典的快捷键。
- Ctrl+D：删除当前行。
- Ctrl+Alt+↓：复制当前行到下一行(复制增加)。
- Ctrl+Alt+↑：复制当前行到上一行(复制增加)。
- Alt+↓：当前行和下面一行交互位置(特别实用，可以省去先剪切，再粘贴了)。
- Alt+↑：当前行和上面一行交互位置(同上)。
- Alt+←：前一个编辑的页面。
- Alt+→：下一个编辑的页面(当然是针对上面那条来说了)。

- Alt+Enter：显示当前选择的资源(例如：工程或文件)的属性。
- Shift+Enter：在当前行的下一行插入空行(这时鼠标可以在当前行的任一位置,不一定是最后)。
- Shift+Ctrl+Enter：在当前行插入空行(原理同上条)。
- Ctrl+Q：定位到最后编辑的地方。
- Ctrl+L：定位在某行 (对于程序超过 100 的人就有福音了)。
- Ctrl+M：最大化当前的 Edit 或 View (再按则反之)。
- Ctrl+/：注释当前行,再按则取消注释。
- Ctrl+O：快速显示 OutLine。
- Ctrl+T ：快速显示当前类的继承结构。
- Ctrl+W：关闭当前 Editer。
- Ctrl+K：参照选中的 Word 快速定位到下一个。
- Ctrl+E：快速显示当前 Editer 的下拉列表(如果当前页面没有显示的用黑体表示)。
- Ctrl+/(小键盘)：折叠当前类中的所有代码。
- Ctrl+×(小键盘)：展开当前类中的所有代码。
- Ctrl+Space：代码助手完成一些代码的插入(但一般和输入法有冲突，可以修改输入法的热键,也可以暂用 Alt+/来代替)。
- Ctrl+Shift+E：显示管理当前打开的所有的 View 的管理器(可以选择关闭，激活等操作)。
- Ctrl+J：正向增量查找(按 Ctrl+J 后,你所输入的每个字母编辑器都提供快速匹配定位到某个单词，如果没有，则在 stutes line 中显示没有找到了，查一个单词时，特别实用，这个功能 Idea 两年前就有了)。
- Ctrl+Shift+J：反向增量查找(和上条相同，只不过是从后往前查)。
- Ctrl+Shift+F4：关闭所有打开的 Editer。
- Ctrl+Shift+X：把当前选中的文本全部变味小写。
- Ctrl+Shift+Y：把当前选中的文本全部变为小写。
- Ctrl+Shift+F：格式化当前代码。
- Ctrl+Shift+P：定位到对于的匹配符(譬如{}) (从前面定位后面时，光标要在匹配符里面，后面到前面，则反之)。
- Ctrl+Shift+R：Open Resource，通过输入名字过滤来打开文件。

在下面列出的快捷键是在重构工作中比较常用的(注：一般重构的快捷键都是 Alt+Shift 开头的)，读者可以借鉴以提高工作效率。

- Alt+Shift+R：重命名，对变量和类的重命名来说，比手工方法能节省很多劳动力。
- Alt+Shift+M：抽取方法 (这是重构里面最常用的方法之一了，尤其是对一大堆泥团代码有用)。
- Alt+Shift+C：修改函数结构(比较实用，有 N 个函数调用了这个方法，修改一次搞定)。
- Alt+Shift+L：抽取本地变量(可以直接把一些魔法数字和字符串抽取成一个变量，

尤其是多处调用的时候)。

- Alt+Shift+F：把 Class 中的 local 变量变为 field 变量(比较实用的功能)。
- Alt+Shift+I：合并变量(可能这样说有点不妥 Inline)。
- Alt+Shift+V：移动函数和变量(不怎么常用)。
- Alt+Shift+Z：重构的后悔药(Undo)。

10.5.2 启动运行速度调优

JVM 提供了各种用于调整内存分配和垃圾回收行为的标准开关和非标准开关。其中一些设置可以提高 JAVA IDE 的性能。由于-X(尤其是 -XX JVM)开关通常是 JVM 或 JVM 供应商特定的，本部分介绍的开关可用于 Sun Microsystems J2SE 1.4.2。

以下设置在大多数系统上将产生比工厂更好的设置性能。

- -vmargs：表示将后面的所有参数直接传递到所指示的 Java VM。
- -Xverify:none：此开关关闭 Java 字节码验证，从而加快了类装入的速度，并使得在仅为验证目的而启动的过程中无须装入类。此开关缩短了启动时间，因此没有理由不使用它。
- -Xms24m：此设置指示 Java 虚拟机将其初始堆大小设置为 24MB。通过指示 JVM 最初应分配给堆的内存数量，可以使 JVM 不必在 IDE 占用较多内存时增加堆大小。
- -Xmx96m：此设置指定 Java 虚拟机应对堆使用的最大内存数量。为此数量设置上限表示 Java 进程消耗的内存数量不得超过可用的物理内存数量。对于具有更多内存的系统可以增加此限制，96 MB 设置有助于确保 IDE 在内存量为 128MB 到 256MB 的系统上能够可靠地执行操作。注意：不要将该值设置为接近或大于系统的物理内存量，否则将在主要回收过程中导致频繁的交换操作。
- -XX:PermSize=20m：此 JVM 开关不仅功能更为强大，而且能够缩短启动时间。该设置用于调整内存“永久区域”(类保存在该区域中)的大小。因此我们向 JVM 提示它将需要的内存量。该设置消除了许多系统启动过程中的主要垃圾收集事件。SunONE Studio 或其他包含更多模块的 IDE 的用户可能希望将该数值设置得更高。

下面列出了其他一些可能对 ECLIPSE 在某些系统(不是所有系统)上的性能产生轻微或明显影响的 JVM 开关。尽管使用它们会产生一定的影响，但仍值得一试。

- -XX:CompileThreshold 100：此开关将降低启动速度，原因是与不使用此开关相比，HotSpot 能够更快地将更多的方法编译为本地代码。其结果是提高了 IDE 运行时的性能，这是因为更多的 UI 代码将被编译而不是被解释。该值表示方法在被编译前必须被调用的次数。
- -XX:+UseConcMarkSweepGC -XX:+UseParNewGC：如果垃圾回收频繁中断，则请尝试使用这些开关。此开关导致 JVM 对主要垃圾回收事件(如果在多处理器工作站上运行，则也适用于次要回收事件)使用不同的算法，这些算法不会影响整个垃圾回收进程。注意：目前尚不确定此收集器是提高还是降低单处理器计算机的

性能。

- -XX:+UseParallelGC：某些测试表明，至少在内存配置相当良好的单处理器系统中，使用此回收算法可以将次要垃圾回收的持续时间减半。注意，这是一个矛盾的问题，事实上此回收器主要适用于具有千兆字节堆的多处理器。尚无可用数据表明它对主要垃圾回收的影响。注意：此回收器与 -XX:+UseConcMarkSweepGC 是互斥的。

例如 512MB 内存的启动参数是：

```
eclipse.exe -vmargs -Xverify:none -Xms64M -Xmx256M -XX:PermSize=20M -
XX:+UseParallelGC
```

1G 内存的启动参数是：

```
eclipse.exe -vmargs -Xverify:none -Xms128M -Xmx512M -XX:PermSize=64M -
XX:MaxPermSize=128M -XX:+UseParallelGC
```

10.5.3　调优前的程序运行状态

笔者使用 Eclipse 3.5 作为日常工作中的主要 IDE 工具，由于安装的插件比较大(如 Klocwork、ClearCase LT 等)、代码也很多，启动 Eclipse 直到所有项目编译完成需要四五分钟。一直对开发环境的速度感到不满意，趁着编写这本书的机会，决定对 Eclipse 进行“动刀”调优。

笔者机器的 Eclipse 运行平台是 32 位 Windows 7 系统，虚拟机为 HotSpot VM 1.5 b64，硬件为 ThinkPad X201、Intel I5 CPU、4GB 物理内存。在初始的配置文件 eclipse.ini 中，除了指定 JDK 的路径，设置最大堆为 512MB 及开启了 JMX 管理(需要在 VisualVM 中收集原始数据)外，未作任何改动，原始配置内容如代码清单 10-2 所示。

代码清单 10-2　Eclipse 3.5 初始配置

```
-vm
D:/ DevSpace/j dkl.5. O/bin/j avaw. exe
- startup
- -launcher.librarye.equinox.launcher-1.o.201.R35X V20090715.jar
 一launCher.library
plugins/org.eclipse. equinox .launcher. win32. win32 .x8 6 1.0.200.v20090519
-product
org. eclipse .epp .package.jee.product
- -launcher.XXMaxPermSize
256M
-showsplash
org.eclipse.platform
-vmargs
- Dosgi.requiredjavaVersion=1.5
-Xmx512m
- Dc am.sun.managemant.j mxr emote
```

为了与调优后的结果进行量化对比，调优开始前笔者先做了一次初始数据测试。测试用例很简单，就是收集从 Eclipse 启动开始，直到所有插件加载完成为止的总耗时及运行状态数据，虚拟机的运行数据通过 VisualVM 及其扩展插件 VisualGC 进行采集。测试过程

中反复启动 Eclipse 数次直到测试结果稳定后，取最后一次运行的结果作为数据样本(为了避免操作系统未能及时进行磁盘缓存而产生的影响)。

Eclipse 启动的总耗时没有办法从监控工具中直接获得，因为 VisualVM 不可能知道 Eclipse 运行到什么阶段才算是启动完成。为了保证测试的准确性，笔者写了一个简单的 Eclipse 插件，用于统计 Eclipse 的启动耗时。由于代码很简单，并且本书不是 Eclispe RCP 的开发教程，所以只列出代码清单 10-3 供读者参考，不再延伸讲解。如果读者需要这个插件，可以使用下面的代码自己编译或者发 E-mail 给笔者索取。

代码清单 10-3　Eclipse 启动耗时统计插件

```
ShowTime.java 代码:
import org. eclipse.j face. dialogs.MessageDialog;
import org.eclipse.swt.widgets.Display;
import org. eclipse. swt.widgets. Shell;
import org.eclipse.ui.IStartup;
public class ShowTime implements IStartup{
public void earlyStartup()  {
Display.getDefault().syncExec (new  Runnable()  f
public void run()  {
long eclipseStartTime=Long.parseLong(System.getProperty("eclipse.
startTime”))j
long costTime=System.currentTimeMillis()  一eclipseStartTime;
Shell shell=Display.getDefault().getActiveShell();
String message="Eclipse 启动耗时: "+costTime+"ms";
MessageDialog.openInformation(shell,  "Information", message);
)
});
)
)
```

plugin. xml 代码：

```
<?xml  version= "1.0”encoding= "UTF-8”?>
<?eclipse version=J'3.4"?>
<plugin>
<extension point=”org. eclipse.ui.startup”>
<startup class="aclipsestarttime. actions. ShowTime"/>
</extension>
</plugin>
```

上述代码打包成 jar 后放到 Eclipse 的 plugins 目录中，反复启动几次后，插件显示的平均时间稳定在 15 秒左右。根据 VisualGC 和 Eclipse 插件收集到的信息，总结原始配置下的测试结果如下：

- 整个启动过程平均耗时约 15 秒。
- 最后一次启动的数据样本中，垃圾收集总耗时 4.149 秒，其中 Full GC 被触发了 19 次，共耗时 3.166 秒。而 Minor GC 被触发了 378 次，共耗时 0.983 秒。
- 加载类 9115 个，耗时 4.114 秒。
- JIT 的编译时间为 1.999 秒。
- 虚拟机 512MB 的堆内存被分配为 40MB 的新生代(31.5MB 的 Eden 空间和 2 个

4MB 的 Surviver 空间)及 472MB 的老年代。

客观地说，由于机器硬件还不错(请读者以 2010 年普通 PC 的标准来衡量)，15 秒的启动时间其实还在可接受的范围以内，但是从 VisualGC 中反映的数据来看，主要问题是非用户程序时间(图 10-2 中的 Compile Time、Class Loader Time、GC Time)非常高，占了整个启动过程耗时的一半以上(这里存在少许夸张成分，因为如果 JIT 编译等动作是在后台线程完成的，用户程序在此期间也正常执行，所以并没有占用一半以上的绝对时间)。虚拟机后台占用太多时间也直接导致 Eclipse 在启动后的使用过程中经常有停顿的感觉，所以进行调优有较大的价值。

Java

第 11 章

虚拟机类的加载机制

前面讲解了 Class 文件存储格式的具体细节，在 Class 文件中描述的各种信息最终都需要加载到虚拟机中之后才能被运行和使用。本章将讲解虚拟机如何加载这些 Class 文件，介绍 Class 文件中的信息进入到虚拟机后会发生什么变化等内容，为读者学习后面的知识打下基础。

11.1 虚拟机类的加载

虚拟机把描述类的数据从 Class 文件加载到内存，并对数据进行校验、转换解析和初始化，最终形成可以被虚拟机直接使用的 Java 类型，这就是虚拟机的类加载机制。与那些在编译时需要进行连接工作的语言不同，在 Java 语言里面，类型的加载和连接过程都是在程序运行期间完成的，这样会在类加载时稍微增加一些性能开销，但是却能为 Java 应用程序提供高度的灵活性，Java 中天生可以动态扩展的语言特性就是依赖运行期动态加载和动态连接这个特点实现的。例如，如果编写一个使用接口的应用程序，可以等到运行时再指定其实际的实现。这种组装应用程序的方式广泛应用于 Java 程序之中。

为了避免语言表达中可能产生的偏差，在本章正式开始之前，笔者先设立如下两个语言上的约定。

(1) 在实际情况中，每个 Class 文件都有可能代表 Java 语言中的一个类或接口，后文中直接对“类”的描述都包括了类和接口的可能性，而对于类和接口需要分开描述的场景会特别指明。

(2) 笔者所说的“Class 文件并非指 Class 必须是存在于具体磁盘中的某个文件，这里说的 Class 文件指的是一串二进制的字节流，无论以何种形式存在都可以。

类从被加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期包括：加载(Loading)、验证(Verification)、准备(Preparation)、解析(Resolution)、初始化(Initialization)、使用(Using)和卸载(Unloading)7 个阶段。其中验证、准备和解析三个部分统称为连接(Linking)。在这 7 个阶段中，加载、验证、准备、初始化和卸载这 5 个阶段的顺序是确定的，类的加载过程必须按照这种顺序按部就班地开始。而解析阶段则不一定，它在某些情况下可以在初始化阶段之后再开始，这是为了支持 Java 语言的运行时绑定(也称为动态绑定或晚期绑定)。请注意这里写的是按部就班地“开始”，而不是按部就班地“进行”或“完成”。因为这些阶段通常都是互相交叉地混合式进行的，通常会在一个阶段执行的过程中调用或激活另外一个阶段。

什么情况下需要开始类加载过程的第一个阶段：加载。虚拟机规范中并没有进行强制约束，这点可以交给虚拟机的具体实现来自由把握。但是对于初始化阶段，虚拟机规范则是严格规定了有且只有 4 种情况必须立即对类进行“初始化”。而加载、验证、准备自然需要在此之前开始：

(1) 遇到 new、getstatic、putstatic 或 invokestatic 这 4 条字节码指令时，如果类没有进行过初始化，则需要先触发其初始化。生成这 4 条指令的最常见的 Java 代码场景是：使用 new 关键字实例化对象的时候、读取或设置一个类的静态字段(被 finaL 修饰、已在编译期把结果放入常量池的静态字段除外)的时候，以及调用一个类的静态方法的时候。

(2) 使用 java.lang.refiect 包的方法对类进行反射调用的时候，如果类没有进行过初始化，则需要先触发其初始化。

(3) 当初始化一个类的时候，如果发现其父类还没有进行过初始化，则需要先触发其

父类的初始化。

(4) 当虚拟机启动时，用户需要指定一个要执行的主类(包含 main()方法的那个类)，虚拟机会先初始化这个主类。

对于这 4 种会触发类进行初始化的场景，虚拟机规范中使用了一个很强烈的限定语："有且只有"，这 4 种场景中的行为称为对一个类进行主动引用。除此之外所有引用类的方式，都不会触发初始化，称为被动引用。下面通过三段代码分别来说明被动引用的过程。

第一段代码：

```
public class SuperClass {

    static{
        System.out.println("SuperClass init!");
    }

    public static int value=123;

}
public class SubClass extends SuperClass{

    static{
        System.out.println("SubClass init!");
    }
}
public class NotInitialization {

    public static void main(String[] args){

        System.out.println(SubClass.value);

    }
}
```

执行上述代码后，会输出"SuperClass init!"，而不是输出"SubClass init!"。由此可见，对于静态字段而言，只有直接定义这个字段的类才会被初始化，因此通过其子类来引用父类中定义的静态字段，只会触发父类的初始化而不会触发子类的初始化。至于是否要触发子类的加载和验证，在虚拟机规范中并未明确规定，这点取决于虚拟机的具体实现。对于 Sun HotSpot 虚拟机而言，可通过-XX:+TraceClassLoading 参数看到此操作会导致子类的加载。

第二段代码：

```
public class NotInitialization {

    public static void main(String[] args){
        //这里的字节码指令为 newarray
             SuperClass[] sca=new SuperClass[10];
    }
}
```

在上述代码中，复用了第一段代码中的 SuperClass，运行之后发现没有输出

"SuperClass init!"，这说明并没有触发类 org.zzz.class! oading.SuperClass 的初始化阶段。但是这段代码里面触发了另外一个名为"Lorg.zzz.classloading.SuperClass"的类的初始化阶段。对于用户代码来说，这并不是一个合法的类名称，它是一个由虚拟机自动生成的、直接继承于 java.lang.Obj ect 的子类，创建动作由字节码指令 newarray 触发。

这个类代表了一个元素类型为 org.zzz.classloading.SuperClass 的一维数组，数组中应有的属性和方法(用户可直接使用的只有被修饰为 public 的 length 属性和 clone()方法)都实现在这个类里。Java 语言中对数组的访问比 C/C++相对安全，因为这个类包装了数组元素的访问方法，而 C/C++直接翻译为对数组指针的移动。在 Java 语言中，当检查到发生数组越界时会抛出 java.lang.ArrayIndexOutofBoundsException 异常。

第三段代码：

```
public class ConstClass {
    public ConstClass() {
        System.out.println("ConstClass construction");
    }
    public static final String HELLO WORLD = "hello world";
}

/**
 * 常量在编译阶段会存入调用类的常量池中，本质上没有直接引用到定义常量的类，因此不会触发定义常量类的初始化
*/
public class NotInitialization {

    public static void main(String[] args) {
        System.out.println(ConstClass.HELLO WORLD);
//      后台打印：
//      hello world
    }

}
```

运行上述代码之后，也没有输出"ConstClass construction"，这是因为虽然在 Java 源码中引用了 ConstClass 类中的常量 HELLOWORLD，但是在编译阶段将此常量的值"hello world"存储到了 NotInitialization 类的常量池中，对常量 ConstClass.HELLOWORLD 的引用实际都被转化为 NotInitialization 类对自身常量池的引用了。也就是说实际上 NotInitialization 的 Class 文件之中并没有 ConstClass 类的符号引用入口，这两个类在编译成 Class 之后就不存在任何联系了。

接口的加载过程与类加载过程稍有一些不同，针对接口需要做一些特殊说明：接口也有初始化过程，这点与类是一致的，上面的代码都是用静态语句块"static{}"来输出初始化信息的，而接口中不能使用"static{}"语句块，但编译器仍然会为接口生成"<clinit>()"类构造器，用于初始化接口中所定义的成员变量。接口与类真正有所区别的是前面讲述的四种"有且仅有"需要开始初始化场景中的第三种：当一个类在初始化时，要求其父类全部都已经初始化过了，但是一个接口在初始化时，并不要求其父接口全部都完成了初始化，只有在真正使用到父接口的时候(如引用接口中定义的常量)才会初始化。

11.2　类的加载过程

类从加载到虚拟机到卸载，它的整个生命周期包括如下 7 个阶段：

- 加载(Loading)
- 验证(Validation)
- 准备(Preparation)
- 解析(Resolution)
- 初始化(Initialization)
- 使用(Using)
- 卸载(Unloading)

其中，验证、准备和解析部分被称为连接(Linking)，如图 11-1 所示。

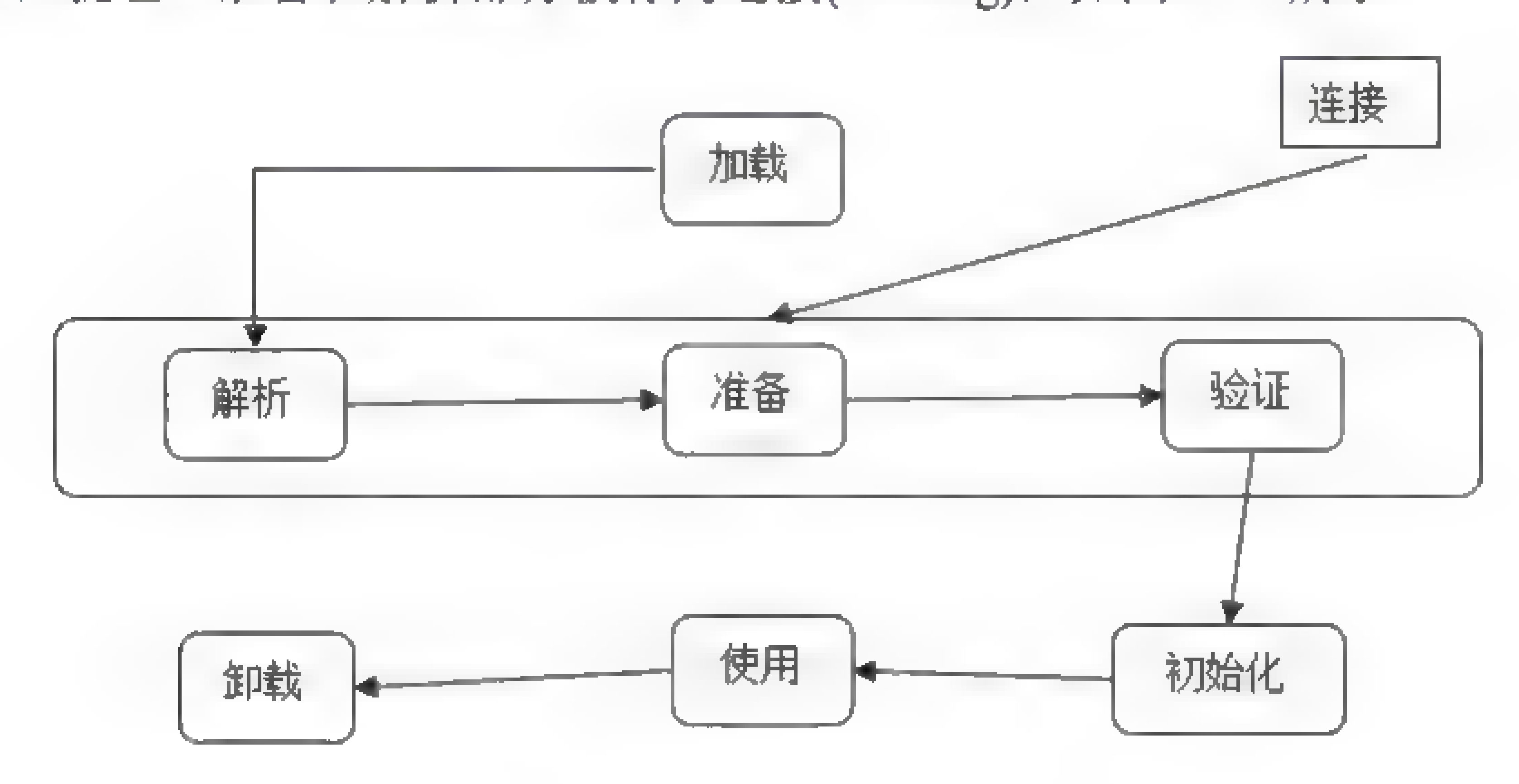


图 11-1　类的加载过程

11.2.1　加载

在加载阶段，虚拟机主要完成三件事：

(1) 通过一个类的全限定名来获取定义此类的二进制字节流。

(2) 将这个字节流所代表的静态存储结构转化为方法区域的运行时数据结构。

(3) 在 Java 堆中生成一个代表这个类的 java.lang.Class 对象，作为方法区域数据的访问入口。

虚拟机规范的这三点要求实际上并不具体，因此虚拟机实现与具体应用的灵活度相当大。例如“通过一个类的全限定名来获取定义此类的二进制字节流”，并没有指明二进制字节流要从一个 Class 文件中获取，准确地说是根本没有指明要从哪里获取及怎样获取。虚拟机设计团队在加载阶段搭建了一个相当开放的、广阔的舞台，Java 发展历程中，充满创造力的开发人员们则在这个舞台上玩出了各种花样，许多举足轻重的 Java 技术都建立在这一基础之上，例如：

- 从 ZIP 包中读取，这很常见，最终成为日后 JAR、EAR、WAR 格式的基础。
- 从网络中获取，这种场景最典型的应用就是 Applet。
- 运行时计算生成，这种场景使用得最多的就是动态代理技术，在 java.lang.reflect.Proxy 中，就是用了 ProxyGenerator.generateProxyClass 来为特定接口生成 *$Proxy 的代理类的二进制字节流。
- 由其他文件生成，典型场景：JSP 应用。
- 从数据库中读取，这种场景相对少见些，有些中间件服务器(如 SAP Netweaver)可以选择把程序安装到数据库中来完成程序代码在集群间的分发。

相对于类加载过程的其他阶段，加载阶段(准确地说，是加载阶段中获取类的二进制字节流的动作)是开发期可控性最强的阶段，因为加载阶段既可以使用系统提供的类加载器来完成，也可以由用户自定义的类加载器去完成，开发人员可以通过定义自己的类加载器去控制字节流的获取方式。

加载阶段完成后，虚拟机外部的二进制字节流就按照虚拟机所需的格式存储在方法区之中，方法区中的数据存储格式由虚拟机实现自行定义，虚拟机规范未规定此区域的具体数据结构。然后在 Java 堆中实例化一个 java.lang.Class 类的对象，这个对象将作为程序访问方法区中的这些类型数据的外部接口。加载阶段与连接阶段的部分内容(如一部分字节码文件格式验证动作)是交叉进行的，加载阶段尚未完成，连接阶段可能已经开始，但这些夹在加载阶段之中进行的动作，仍然属于连接阶段的内容，这两个阶段的开始时间仍然保持着固定的先后顺序。

11.2.2 验证

验证阶段作用是保证 Class 文件的字节流包含的信息符合 JVM 规范，不会给 JVM 造成危害。如果验证失败，就会抛出一个 java.lang.VerifyError 异常或其子类异常。验证是连接阶段的第一步，这一阶段的目的是为了确保 Class 文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。

Java 语言本身是相对安全的语言(依然是相对于 C/C++来说)，使用纯粹的 Java 代码无法做到诸如访问数组边界以外的数据、将一个对象转型为它并未实现的类型、跳转到不存在的代码行之类的事情，如果这样做了，编译器将拒绝编译。但前面已经说过，Class 文件并不一定要求用 Java 源码编译而来，可以使用任何途径，包括用十六进制编辑器直接编写来产生 Class 文件。在字节码的语言层面上，上述 Java 代码无法做到的事情都是可以实现的，至少语义上是可以表达出来的。虚拟机如果不检查输入的字节流，对其完全信任的话，很可能会因为载入了有害的字节流而导致系统崩溃，所以验证是虚拟机对自身保护的一项重要工作。尽管验证阶段是非常重要的，并且验证阶段的工作量在虚拟机的类加载子系统中占了很大一部分，但虚拟机规范对这个阶段的限制和指导显得非常笼统，仅仅说了一句如果验证到输入的字节流不符合 Class 文件的存储格式，就抛出一个 java.lang.VerifyError 异常或其子类异常，具体应当检查哪些方面，如何检查，何时检查，都没有强制要求或明确说明，所以不同的虚拟机对类验证的实现可能会有所不同，但大致上都会完成下面四个阶段的检验过程：文件格式验证、元数据验证、字节码验证和符号引

用验证。

(1) 文件格式验证：验证字节流文件是否符合 Class 文件格式的规范，并且能被当前虚拟机正确的处理。

(2) 元数据验证：是对字节码描述的信息进行语义分析，以保证其描述的信息符合 Java 语言的规范。

(3) 字节码验证：主要是进行数据流和控制流的分析，保证被校验类的方法在运行时不会危害虚拟机。

(4) 符号引用验证：符号引用验证发生在虚拟机将符号引用转化为直接引用的时候，这个转化动作将在解析阶段中发生。

在接下来的内容中，将详细讲解上述 4 个验证阶段的基本知识。

1．文件格式验证

第一阶段要验证字节流是否符合 Class 文件格式的规范，并且能被当前版本的虚拟机处理。这一阶段可能包括下面这些验证点。

- 是否以魔数 OxCAFEBABE 开头。
- 主、次版本号是否在当前虚拟机处理范围之内。
- 常量池的常量中是否有不被支持的常量类型(检查常量 tag 标志)。
- 指向常量的各种索引值中是否有指向不存在的常量或不符合类型的常量。
- CONSTANT_Utf8_info 型的常量中是否有不符合 UTF8 编码的数据。
- Class 文件中各个部分及文件本身是否有被删除的或附加的其他信息。

实际上第一阶段的验证点还远不止这些，上面这些只是从 HotSpot 虚拟机源码中摘抄的一小部分，该验证阶段的主要目的是保证输入的字节流能正确地解析并存储于方法区之内，格式上符合描述一个 Java 类型信息的要求。这阶段的验证是基于字节流进行的，经过了这个阶段的验证之后，字节流才会进入内存的方法区中进行存储，所以后面的三个验证阶段全部是基于方法区的存储结构进行的。

2．元数据验证

第二阶段是对字节码描述的信息进行语义分析，以保证其描述的信息符合 Java 语言规范的要求，这个阶段可能包括的验证点如下：

- 这个类是否有父类(除了 java.lang.Object 之外，所有的类都应当有父类)。
- 这个类的父类是否继承了不允许被继承的类(被 final 修饰的类)。
- 如果这个类不是抽象类，是否实现了其父类或接口之中要求实现的所有方法。
- 类中的字段、方法是否与父类产生了矛盾(例如覆盖了父类的 final 字段，或者出现不符合规则的方法重载，例如方法参数都一致，但返回值类型却不同等)。

第二阶段的主要目的是对类的元数据信息进行语义校验，保证不存在不符合 Java 语言规范的元数据信息。

3．字节码验证

第三阶段是整个验证过程中最复杂的一个阶段，主要工作是进行数据流和控制流分

析。在第二阶段对元数据信息中的数据类型做完校验后，这阶段将对类的方法体进行校验分析。这阶段的任务是保证被校验类的方法在运行时不会做出危害虚拟机安全的行为，例如：

- 保证任意时刻操作数栈的数据类型与指令代码序列都能配合工作，例如不会出现类似这样的情况：在操作栈中放置了一个 int 类型的数据，使用时却按 long 类型来加载入本地变量表中。
- 保证跳转指令不会跳转到方法体以外的字节码指令上。
- 保证方法体中的类型转换是有效的，例如可以把一个子类对象赋值给父类数据类型，这是安全的，但是把父类对象赋值给子类数据类型，甚至把对象赋值给予它毫无继承关系、完全不相干的一个数据类型，则是危险和不合法的。

如果一个类方法体的字节码没有通过字节码验证，那肯定是有问题的；但如果一个方法体通过了字节码验证，也不能说明其一定就是安全的。即使字节码验证之中进行了大量的检查，也不能保证这一点。这里涉及了离散数学中一个很著名的问题“Halting Problem”：通俗一点的说法就是，通过程序去校验程序逻辑是无法做到绝对准确的——不能通过程序准确地检查出程序是否能在有限的时间之内结束运行。

由于数据流验证的高复杂性，虚拟机设计团队为了避免将过多的时间消耗在字节码验证阶段，在 JDK 1.6 之后的 Javac 编译器中进行了一种优化，给方法体的 Code 属性的属性表中增加了一项名为“StackMapTable”的属性，这项属性描述了方法体中所有的基本块(Basic Block，按照控制流拆分的代码块)开始时本地变量表和操作栈应有的状态，这可以将字节码验证的类型推导转变为类型检查从而节省一些时间。当然，理论上 StackMapTable 属性也存在错误或被篡改的可能，所以是否有可能在恶意篡改了 Code 属性的同时，也生成相应的 StackMapTable 属性来骗过虚拟机的类型校验则是虚拟机实现时值得思考的问题。

在 JDK 1.6 以后的版本的 HotSpot 虚拟机中，提供了-XX:-UseSplitVerifier 选项来关闭掉这项优化，或者使用参数-XX:+FailOverToOldVerifier 要求在类型校验失败的时候退回到旧的类型推导方式进行校验。而在 JDK 1.7 之后，对于主版本号大于 50 的 Class 文件，使用类型检查来完成数据流分析校验则是唯一的选择，不允许再退回到类型推导的校验方式。

4．符号引用验证

最后一个阶段的校验发生在虚拟机将符号引用转化为直接引用的时候，这个转化动作将在连接的第三个阶段——解析阶段中发生。符号引用验证可以看做是对类自身以外(常量池中的各种符号引用)的信息进行匹配性的校验，通常需要校验以下内容：

- 符号引用中通过字符串描述的全限定名是否能找到对应的类。
- 在指定类中是否存在符合方法的字段描述符及简单名称所描述的方法和字段。
- 符号引用中的类、字段和方法的访问性(private、protected、public、default)是否可被当前类访问。

符号引用验证的目的是确保解析动作能正常执行，如果无法通过符号引用验证，将会抛出一个 java.lang.IncompatibleClassChangeError 异常的子类，如 java.lang.IllegalAccessError、

java.lang.NoSuchFieldError、java.lang.NoSuchMethodError 等。

验证阶段对于虚拟机的类加载机制来说，是一个非常重要的、但不一定是必要的阶段。如果所运行的全部代码(包括自己写的、第三方包中的代码)都已经被反复使用和验证过，在实施阶段就可以考虑使用-Xverify:none 参数来关闭大部分的类验证措施，以缩短虚拟机类加载的时间。

11.2.3 准备

准备阶段是正式为类变量分配内存并设置类变量初始值的阶段，这些内存都将在方法区中进行分配。这个时候内存分配的仅包括类变量(static 变量)，不包括实例变量，实例变量将会在对象实例化时随着对象一起分配在 java 堆中。其次是这里所说的初始值“通常情况下”是数据类型的零值(随后在初始化阶段生成定义的初值)。如果该变量被 final 修饰，将在编译时生成 ConstantValue，这样在准备阶段将直接设置成该初值。

准备阶段是正式为类变量分配内存并设置类变量初始值的阶段，这些内存都将在方法区中进行分配。这个阶段中有两个容易产生混淆的概念需要强调一下，首先是这时候进行内存分配的仅包括类变量(被 static 修饰的变量)，而不包括实例变量，实例变量将会在对象实例化时随着对象一起分配在 Java 堆中。其次是这里所说的初始值“通常情况”下是数据类型的零值，假设一个类变量的定义为：

```
public static int value=123;
```

那么变量 value 在准备阶段过后的初始值为 0 而不是 123，因为这时候尚未开始执行任何 Java 方法，而把 value 赋值为 123 的 putstatic 指令是程序被编译后，存放于类构造器<clinit>()方法之中，所以把 value 赋值为 123 的动作将在初始化阶段才会被执行。在“通常情况”下初始值是零值，那相对的会有一些“特殊情况”：如果类字段的字段属性表中存在 ConstantValue 属性，那在准备阶段变量 value 就会被初始化为 ConstantValue 属性所指定的值，假设上面类变量 value 的定义变为：

```
public static final int value=123;
```

编译时 Javac 将会为 value 生成 ConstantValue 属性，在准备阶段虚拟机就会根据 ConstantValue 的设置将 value 赋值为 123。

11.2.4 解析

解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程，符号引用在本书前面讲解 Class 文件格式的时候就已经出现过多次，在 Class 文件中它以 CONSTANT_Class_info、CONSTANT_Fieldref_info、CONSTANT—Methodref_info 等类型的常量出现，那解析阶段中所说的直接引用与符号引用又有什么关联呢？

- 符号引用(Symbolic References)：符号引用以一组符号来描述所引用的目标，符号可以是任何形式的字面量，只要使用时能无歧义地定位到目标即可。符号引用与虚拟机实现的内存布局无关，引用的目标并不一定已经加载到内存中。
- 直接引用(Direct References)：直接引用可以是直接指向目标的指针、相对偏移量

或是一个能间接定位到目标的句柄。直接引用是与虚拟机实现的内存布局相关的，同一个符号引用在不同虚拟机实例上翻译出来的直接引用一般不会相同。如果有了直接引用，那引用的目标必定已经在内存中存在。

虚拟机规范之中并未规定解析阶段发生的具体时间，只要求了在执行 anewarray、checkcast、getfield、getstatic、instanceof、invokeinterface、invokespecial、invokestatic、invokevirtual、multianewarray、new、putfield 和 putstatic 这 13 个用于操作符号引用的字节码指令之前，先对它们所使用的符号引用进行解析。所以虚拟机实现会根据需要来判断，到底是在类被加载器加载时就对常量池中的符号引用进行解析，还是等到一个符号引用将要被使用前才去解析它。

对同一个符号引用进行多次解析请求是很常见的事情，虚拟机实现可能会对第一次解析的结果进行缓存(在运行时常量池中记录直接引用，并把常量标识为已解析状态)从而避免解析动作重复进行。无论是否真正执行了多次解析动作，虚拟机需要保证的都是在同一个实体中，如果一个符号引用之前已经被成功解析过，那么后续的引用解析请求就应当一直成功；同样地，如果第一次解析失败了，其他指令对这个符号的解析请求也应该收到相同的异常。

解析动作主要针对类或接口、字段、类方法、接口方法 4 类符号引用进行，分别对应于常量池的 CONSTANT-Class_ info、CONSTANT- Fieldref info、CONSTANT_Methodref_ info 及 CONSTANT_InterfaceMethodref info 四种常量类型。下面将讲解这 4 种引用的解析过程。

1．类或接口的解析

假设当前代码所处的类为 D，如果要把一个从未解析过的符号引用 N 解析为一个类或接口 C 的直接引用，那虚拟机完成整个解析的过程需要包括以下三个步骤。

(1) 如果 C 不是一个数组类型，那虚拟机将会把代表 N 的全限定名传递给 D 的类加载器去加载这个类 C。在加载过程中，由于元数据验证、字节码验证的需要，又将可能触发其他相关类的加载动作，例如加载这个类的父类或实现的接口。一旦这个加载过程出现了任何异常，解析过程就将宣告失败。

(2) 如果 C 是一个数组类型，并且数组的元素类型为对象，也就是 N 的描述符会是类似“Ljava.lang.Integer”的形式，那将会按照第 1 点的规则加载数组元素类型。如果 N 的描述符如前面所假设的形式，需要加载的元素类型就是“java.lang.Integer”，接着由虚拟机生成一个代表此数组维度和元素的数组对象。

(3) 如果上面的步骤没有出现任何异常，那么 C 在虚拟机中实际上已经成为一个有效的类或接口了，但在解析完成之前还要进行符号引用验证，确认 C 是否具备对 D 的访问权限。如果发现不具备访问权限，将抛出 java.lang.IllegalAccessError 异常。

2．字段解析

要解析一个未被解析过的字段符号引用，首先将会对字段表内 class index 项中索引的 CONSTANT Class info 符号引用进行解析，也就是字段所属的类或接口的符号引用。如果在解析这个类或接口符号引用的过程中出现了任何异常，都会导致字段符号引用解析的失

败。如果解析成功完成，那将这个字段所属的类或接口用 C 表示，虚拟机规范要求按照如下步骤对 C 进行后续字段的搜索：

(1) 如果 C 本身就包含了简单名称和字段描述符都与目标相匹配的字段，则返回这个字段的直接引用，查找结束。

(2) 否则，如果在 C 中实现了接口，将会按照继承关系从上往下递归搜索各个接口和它的父接口，如果接口中包含了简单名称和字段描述符都与目标相匹配的字段，则返回这个字段的直接引用，查找结束。

(3) 否则，如果 C 不是 java.lang.Object 的话，将会按照继承关系从上往下递归搜索其父类，如果在父类中包含了简单名称和字段描述符都与目标相匹配的字段，则返回这个字段的直接引用，查找结束。

(4) 否则，查找失败，抛出 java.lang.NoSuchFieldError 异常。

如果查找过程成功返回了引用，将会对这个字段进行权限验证，如果发现不具备对字段的访问权限，将抛出 java.lang.IllegalAccessError 异常。

在实际应用中，虚拟机的编译器实现可能会比上述规范要求得更加严格一些，如果有一个同名字段同时出现在 C 的接口和父类中，或者同时在自己或父类的多个接口中出现，那编译器将可能拒绝编译。例如在下面的演示代码中，如果注释了 Sub 类中的“public static int A=4;”，接口与父类同时存在字段 A，那编译器将提示“The field Sub.A is ambiguous”，并且会拒绝编译这段代码。

```
package org.zzz.classloading j
public class FieldResolution{
interface Interface0  {
int A=0:
}
interface Interfacel extends Interface0  {
int A=1;
}
interface Interface2  {
int A=2;
}
static class Parent implements Interfacel  {
public static int A=3;
}
static class Sub extends Parent implements Interface2  {
public static intA=4;
}
public static void main(String[] args)  {
System.out.println (Sub.A);
}
}
```

3. 类方法解析

类方法解析的第一个步骤与字段解析一样，也是需要先解析出类方法表的 class_index 项中索引的方法所属的类或接口的符号引用，如果解析成功，我们依然用 C 表示这个类，接下来虚拟机将会按照如下步骤进行后续的类方法搜索：

(1) 类方法和接口方法符号引用的常量类型定义是分开的，如果在类方法表中发现 class index 中索引的 C 是个接口，那就直接抛出 java.lang. IncompatibleClassChangeError 异常。

(2) 如果通过了第(1)步，在类 C 中查找是否有简单名称和描述符都与目标相匹配的方法，如果有则返回这个方法的直接引用，查找结束。

(3) 否则，在类 C 的父类中递归查找是否有简单名称和描述符都与目标相匹配的方法，如果有则返回这个方法的直接引用，查找结束。

(4) 否则，在类 C 实现的接口列表及它们的父接口之中递归查找是否有简单名称和描述符都与目标相匹配的方法，如果存在匹配的方法，说明类 C 是一个抽象类，这时候查找结束，抛出 java.lang.AbstractMethodError 异常。

(5) 否则宣告方法查找失败，抛出 java.lang.NoSuchMethodError。

最后，如果查找过程成功返回了直接引用，将会对这个方法进行权限验证：如果发现不具备对此方法的访问权限，将抛出 java.lang.IllegalAccessError 异常。

4. 接口方法解析

接口方法也是需要先解析出接口方法表的 class index 项中索引的方法所属的类或接口的符号引用，如果解析成功，依然用 C 表示这个接口，接下来虚拟机将会按照如下步骤进行后续的接口方法搜索：

(1) 与类方法解析相反，如果在接口方法表中发现 class index 中的索引 C 是个类而不是接口，那就直接抛出 java.lang.IncompatibleClassChangeError 异常。

(2) 否则，在接口 C 中查找是否有简单名称和描述符都与目标相匹配的方法，如果有则返回这个方法的直接引用，查找结束。

(3) 否则，在接口 C 的父接口中递归查找，直到 java.lang.Object 类(查找范围会包括 Obj ect 类)为止，看是否有简单名称和描述符都与目标相匹配的方法，如果有则返回这个方法的直接引用，查找结束。

(4) 否则，宣告方法查找失败，抛出 java.lang. NoSuchMethodError 异常。

由于接口中的所有方法都默认是 public 的，所以不存在访问权限的问题，因此接口方法的符号解析应当不会抛出 java.lang.IllegalAccessError 异常。

11.2.5 初始化

类初始化阶段是类加载过程的最后一步，前面的类加载过程中，除了在加载阶段用户应用程序可以通过自定义类加载器参与之外，其余动作完全由虚拟机主导和控制。到了初始化阶段，才真正开始执行类中定义的 Java 程序代码(或者说是字节码)。

在初始化阶段才真正开始执行类中定义的 Java 程序代码。在准备阶段中，变量已经赋过一次系统要求的初始值，而在初始化阶段，则是根据程序员通过程序制定的计划来赋值。或者说，初始化阶段是执行类构造器<clinit>()方法的过程。

在准备阶段，变量已经赋过一次系统要求的初始值，而在初始化阶段，则是根据程序员通过程序制定的主观计划去初始化类变量和其他资源，或者可以从另外一个角度来表

达：初始化阶段是执行类构造器<clinit>()方法的过程。我们放到后面再讲<clinit>()方法是怎么生成的，在这里，我们先看一下<clinit>()方法执行过程中可能会影响程序运行行为的一些特点和细节，这部分相对更贴近于普通的程序开发人员。

- <clinit>()方法是由编译器自动收集类中的所有类变量的赋值动作和静态语句块(static{}块)中的语句合并产生的，编译器收集的顺序是由语句在源文件中出现的顺序所决定的，静态语句块中只能访问到定义在静态语句块之前的变量，定义在它之后的变量，在前面的静态语句块中可以赋值，但是不能访问。
- <clinit>()方法与类的构造函数(或者说实例构造器<init>()方法)不同，它不需要显式地调用父类构造器，虚拟机会保证在子类的<clinit>()方法执行之前，父类的<clinit>()方法已经执行完毕。因此在虚拟机中第一个被执行的<clinit>()方法的类肯定是 java.lang.Object。
- 由于父类的<clinit>()方法先执行，也就意味着父类中定义的静态语句块要优先于子类的变量赋值操作。
- 虚拟机会保证一个类的<clinit>()方法在多线程环境中被正确地加锁和同步，如果多个线程同时去初始化一个类，那么只会有一个线程去执行这个类的<clinit>()方法，其他线程都需要阻塞等待，直到活动线程执行<clinit>()方法完毕。如果在一个类的<clinit>()方法中有耗时很长的操作，那就可能造成多个进程阻塞，在实际

应用中这种阻塞往往是很隐蔽的，例如下面的代码演示了这种场景。

```
  static class DeadLoopClass {
    static {
           11 4tp果不加上这个if语句,编译器将提示 "Initializer does not complete
normally" .
      if (true) {
        System.out.println(Thread.currentThread() + "init DeadLoopClass");
        while (true) {
                  }
           }
 public static void main(String[J args) {
    Runnable script = new Runnable() {
      public void run() {
           . System.out.println(Thread.currentThread()  + "start") ;
         DeadLoopClass dlc = new DeadLoopClass();
        System.out.println(Thread.currentThread() + " run over");
      };
    Thread threadl = new Thread(script);
    Thread thread2 = new Thread(script);
           threadl.start ( ) ;
       thread2.start () ;
    }
```

运行结果如下，一条线程正在死循环以模拟长时间操作；另外一条线程在阻塞等待：

```
Thread [Thread-0, 5,main] start
Thread [Thread 1,5,main] start
Thread[Thread 0, 5,main] init  DeadLoopClass
```

由此可见，<clinit>()方法的执行顺序是按照声明的顺序排列的。如果类中没有声明任

何类变量和静态初始化语句，那么就不会生成 clinit 语句。如果类中仅有类的 final 常量或者该常量的初始化语句是由编译期常量组成的语句，那么也不会生成 clinit 方法，即只有存在需要执行 java 代码的赋值才会生成 clinit。接下来通过三段代码来说明<clinit>()方法的执行顺序。

第一段：

```
public class StaticParent {

    static int parent time = (int)(Math.random());

    static final int PARENT FINAL = -1;

    static {
        System.out.println(">>>StaticParent 初始化>>");
    }
}
```

第二段：

```
public class StaticChild extends StaticParent{
    //不需要 clinit 函数初始化
    static final int CHILD STATIC 1 = -1;
    //需要 clinit 初始化
    static final int CHILD STATIC 2 = (int)(Math.random()*10);

    //blank final
    static int CHILD STATIC 3;
    //缺省值
    static int CHILD STATIC 4 = 0;

    static StaticChild staticChild = new StaticChild();

    public StaticChild() {
        System.out.println(">>>StaticChild 构造函数>>>");
        CHILD STATIC 3 = 3;
        CHILD STATIC 4 = 4;
    }

    static {
        System.out.println(">>>StaticChild 静态初始化>>>");
        CHILD STATIC 3 = 1;
        CHILD STATIC 4 = 2;
    }

    public static StaticChild getInstance(){
        return staticChild;
    }
}
```

第三段：

```
public class StaticMain {
    public static void main(String []args){
       StaticChild sc = StaticChild.getInstance();
```

```
            System.out.println("CHILD STATIC 1:"+sc.staticChild.CHILD_STATIC_1);
System.out.println("CHILD STATIC 2:"+sc.staticChild.CHILD_STATIC_2);
            System.out.println("CHILD STATIC 3:"+sc.staticChild.CHILD_STATIC_3);
            System.out.println("CHILD STATIC 4:"+sc.staticChild.CHILD_STATIC_4);
    }
}
```

执行后会输出：

```
>>>StaticParent 初始化>>
>>>StaticChild 构造函数>>>
>>>StaticChild 静态初始化>>>
CHILD_STATIC_1:-1
CHILD_STATIC_2:3
CHILD_STATIC_3:1
CHILD_STATIC_4:2
```

通过上述执行结果，可以看出<clinit>()方法的执行顺序。

11.3 类加载器

类加载器是沙箱的第一道防线，因为代码都是由它装入到 JVM 中的，在其中也有可能包括危险的代码。类加载器的安全作用有如下三点：

- 保护善意代码不受恶意代码的干扰。
- 保护已验证的类库。
- 代码放入有不同的行为限制的各个保护域中。

类加载体系通过使用不同的类加载器把类放入不同的名字空间中从而保护善意代码不受恶意代码的干扰。JVM 为每个类加载器维护一个名字空间。例如，如果 JVM 在某个名字空间中加载了一个称为 volcano 的类，就不能再在这个名字空间中加载另一个也称为 volcano 的类，除非你再创建另一个名字空间。也就是说，如果 JVM 有三个名字空间，你就可以加载三个叫做 volcano 的类，一个名字空间一个。

11.3.1 类加载器的基础知识

类加载器是 Java 语言的一个创新，也是 Java 语言流行的重要原因之一。它使得 Java 类可以被动态加载到 Java 虚拟机中并执行。类加载器从 JDK 1.0 就开始出现了，最初是为了满足 Java Applet 的需要而开发出来的。Java Applet 需要从远程下载 Java 类文件到浏览器中并执行。现在类加载器在 Web 容器和 OSGi 中得到了广泛的使用。一般来说，Java 应用的开发人员不需要直接同类加载器进行交互。Java 虚拟机默认的行为就已经足够满足大多数情况的需求了。不过如果遇到了需要与类加载器进行交互的情况，而对类加载器的机制又不是很了解的话，就很容易花大量的时间去调试 ClassNotFoundException 和 NoClassDefFoundError 等异常。

类加载器(Class Loader)用来加载 Java 类到 Java 虚拟机中。一般来说，Java 虚拟机使用 Java 类的方式：Java 源程序(.java 文件)在经过 Java 编译器编译之后就被转换成

Java 字节代码(.class 文件)。类加载器负责读取 Java 字节代码，并转换成 java.lang.Class 类的一个实例。每个这样的实例用来表示一个 Java 类。通过此实例的 newInstance()方法就可以创建出该类的一个对象。实际的情况可能更加复杂，比如 Java 字节代码可能是通过工具动态生成的，也可能是通过网络下载的。

基本上所有的类加载器都是 java.lang.ClassLoader 类的一个实例。下面详细介绍这个 Java 类。

1. java.lang.ClassLoader 类介绍

类 java.lang.ClassLoader 的基本职责就是根据一个指定的类的名称，找到或者生成其对应的字节代码，然后从这些字节代码中定义出一个 Java 类，即 java.lang.Class 类的一个实例。除此之外，ClassLoader 还负责加载 Java 应用所需的资源，如图像文件和配置文件等。不过本文只讨论其加载类的功能。为了完成加载类的这个职责，ClassLoader 提供了一系列的方法，比较重要的方法如表 11-1 所示。

表 11-1　ClassLoader 中与加载类相关的方法

方　法	说　明
getParent()	返回该类加载器的父类加载器
loadClass(String name)	加载名称为 name 的类，返回的结果是 java.lang.Class 类的实例
findClass(String name)	查找名称为 name 的类，返回的结果是 java.lang.Class 类的实例
findLoadedClass(String name)	查找名称为 name 的已经被加载过的类，返回的结果是 java.lang.Class 类的实例
defineClass(String name, byte[] b, int off, int len)	把字节数组 b 中的内容转换成 Java 类，返回的结果是 java.lang.Class 类的实例。这个方法被声明为 final 的
resolveClass(Class<?> c)	链接指定的 Java 类

对于表 11-1 中给出的方法，表示类名称的 name 参数的值是类的二进制名称。需要注意的是内部类的表示，如 com.example.Sample$1 和 com.example.Sample$Inner 等表示方式。这些方法会在下面介绍类加载器的工作机制时，做进一步的说明。下面介绍类加载器的树状组织结构。

2. 类加载器的树状组织结构

Java 中的类加载器大致可以分成两类，一类是系统提供的，另外一类则是由 Java 应用开发人员编写的。系统提供的类加载器主要有下面三个：

- 引导类加载器(Bootstrap Class Loader)：它用来加载 Java 的核心库，是用原生代码来实现的，并不继承自 java.lang.ClassLoader。
- 扩展类加载器(Extensions Class Loader)：它用来加载 Java 的扩展库。Java 虚拟机的实现会提供一个扩展库目录。该类加载器在此目录里面查找并加载 Java 类。
- 系统类加载器(System Class Loader)：它根据 Java 应用的类路径(CLASSPATH)来加载 Java 类。一般来说，Java 应用的类都是由它来完成加载的。可以通过

ClassLoader.getSystemClassLoader()来获取它。

除了系统提供的类加载器以外，开发人员可以通过继承 java.lang.ClassLoader 类的方式实现自己的类加载器，以满足一些特殊的需求。

除了引导类加载器之外，所有的类加载器都有一个父类加载器。通过表 11-1 中给出的 getParent()方法可以得到。对于系统提供的类加载器来说，系统类加载器的父类加载器是扩展类加载器，而扩展类加载器的父类加载器是引导类加载器；对于开发人员编写的类加载器来说，其父类加载器是加载此类加载器 Java 类的类加载器。因为类加载器 Java 类如同其他的 Java 类一样，也是要由类加载器来加载的。一般来说，开发人员编写的类加载器的父类加载器是系统类加载器。类加载器通过这种方式组织起来，形成树状结构。树的根节点就是引导类加载器。图 11-2 中给出了一个典型的类加载器树状组织结构示意图，其中的箭头指向的是父类加载器。

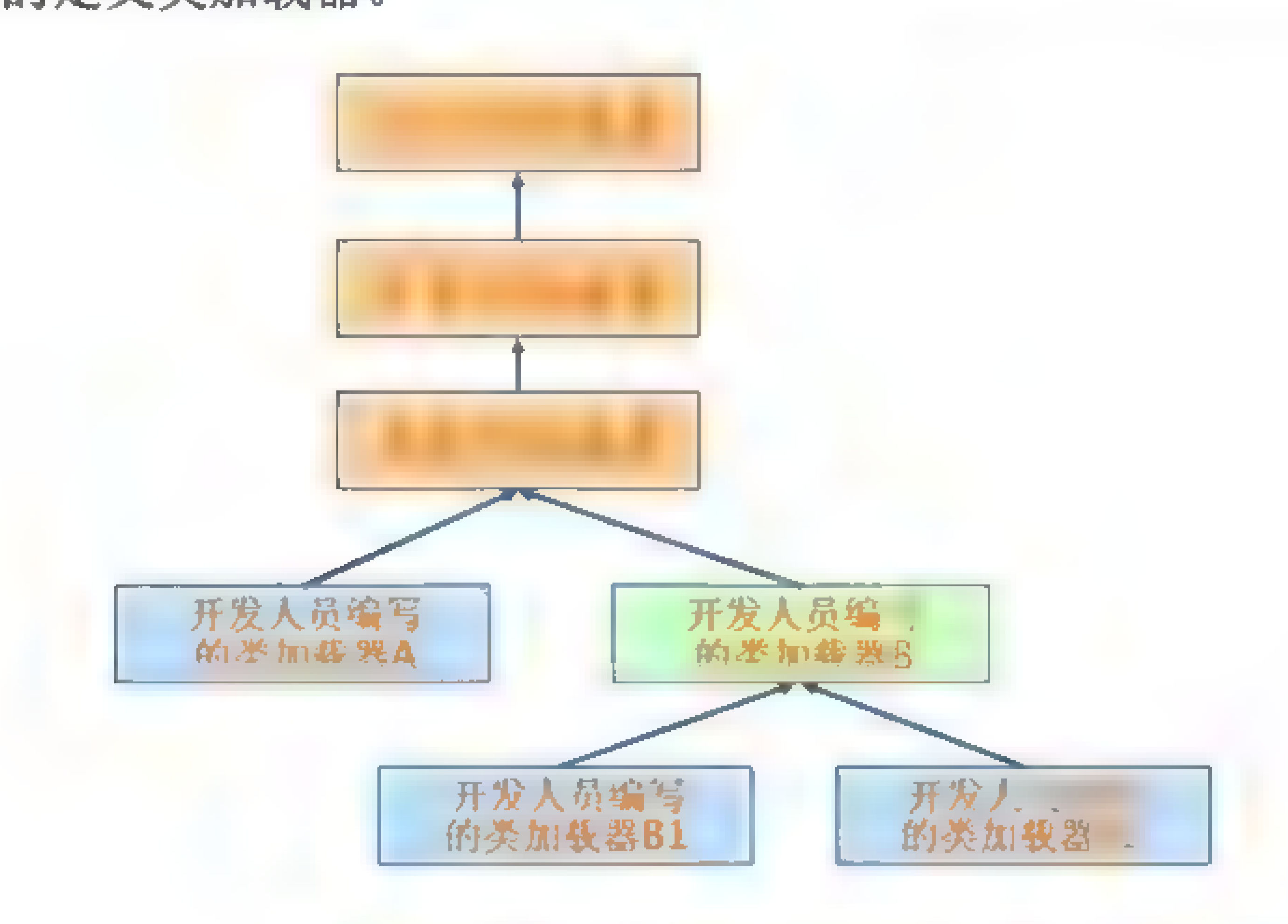


图 11-2　类加载器树状组织结构示意图

例如下面的代码演示了类加载器的树状组织结构。

```
public class ClassLoaderTree {
    public static void main(String[] args) {
        ClassLoader loader = ClassLoaderTree.class.getClassLoader();
        while (loader != null) {
            System.out.println(loader.toString());
            loader = loader.getParent();
        }
    }
}
```

每个 Java 类都维护着一个指向定义它的类加载器的引用，通过 getClassLoader()方法就可以获取到此引用。代码清单 1 中通过递归调用 getParent()方法来输出全部的父类加载器。上述代码的运行结果如下所示。

```
sun.misc.Launcher$AppClassLoader@9304b1
sun.misc.Launcher$ExtClassLoader@190d11
```

如上述输出结果所示，第一个输出的是 ClassLoaderTree 类的类加载器，即系统类加

载器。它是 sun.misc.Launcher$AppClassLoader 类的实例；第二个输出的是扩展类加载器，是 sun.misc.Launcher$ExtClassLoader 类的实例。需要注意的是这里并没有输出引导类加载器，这是由于有些 JDK 的实现对于父类加载器是引导类加载器的情况，getParent()方法返回 null。

在了解了类加载器的树状组织结构之后，下面介绍类加载器的代理模式。

3. 类加载器的代理模式

类加载器在尝试自己去查找某个类的字节代码并定义它时，会先代理给其父类加载器，由父类加载器先去尝试加载这个类，依次类推。在介绍代理模式背后的动机之前，首先需要说明一下 Java 虚拟机是如何判定两个 Java 类是相同的。Java 虚拟机不仅要看类的全名是否相同，还要看加载此类的类加载器是否一样。只有两者都相同的情况，才认为两个类是相同的。即便是同样的字节代码，被不同的类加载器加载之后所得到的类，也是不同的。比如一个 Java 类 com.example.Sample，编译之后生成了字节代码文件 Sample.class。两个不同的类加载器 ClassLoaderA 和 ClassLoaderB 分别读取了这个 Sample.class 文件，并定义出两个 java.lang.Class 类的实例来表示这个类。这两个实例是不相同的。对于 Java 虚拟机来说，它们是不同的类。试图对这两个类的对象进行相互赋值，会抛出运行时异常 ClassCastException。下面通过示例来具体说明。例如在下面的代码中，给出了 Java 类 com.example.Sample。

```
package com.example;
public class Sample {
   private Sample instance;
   public void setSample(Object instance) {
      this.instance = (Sample) instance;
   }
}
```

在上述代码中，类 com.example.Sample 的方法 setSample 接受一个 java.lang.Object 类型的参数，并且会把该参数强制转换成 com.example.Sample 类型。例如测试 Java 类是否相同的代码如下。

```
public void testClassIdentity() {
    String classDataRootPath = "C:\\workspace\\Classloader\\classData";
    FileSystemClassLoader fscl1 = new
FileSystemClassLoader(classDataRootPath);
    FileSystemClassLoader fscl2 = new
FileSystemClassLoader(classDataRootPath);
    String className = "com.example.Sample";
    try {
        Class<?> class1 = fscl1.loadClass(className);
        Object obj1 = class1.newInstance();
        Class<?> class2 = fscl2.loadClass(className);
        Object obj2 = class2.newInstance();
        Method setSampleMethod = class1.getMethod("setSample",
java.lang.Object.class);
        setSampleMethod.invoke(obj1, obj2);
    } catch (Exception e) {
```

```
        e.printStackTrace();
    }
}
```

在上述代码中，使用了类 FileSystemClassLoader 的两个不同实例来分别加载类 com.example.Sample，得到了两个不同的 java.lang.Class 的实例，接着通过 newInstance() 方法分别生成了两个类的对象 obj1 和 obj2，最后通过 Java 的反射 API 在对象 obj1 上调用方法 setSample，试图把对象 obj2 赋值给 obj1 内部的 instance 对象。上述代码的运行结果如下。

```
java.lang.reflect.InvocationTargetException
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl
.java:25)
at java.lang.reflect.Method.invoke(Method.java:597)
at classloader.ClassIdentity.testClassIdentity(ClassIdentity.java:26)
at classloader.ClassIdentity.main(ClassIdentity.java:9)
Caused by: java.lang.ClassCastException: com.example.Sample
cannot be cast to com.example.Sample
at com.example.Sample.setSample(Sample.java:7)
... 6 more
```

从上述运行结果可以看到，运行时抛出了 java.lang.ClassCastException 异常。虽然两个对象 obj1 和 obj2 的类的名字相同，但是这两个类是由不同的类加载器实例来加载的，因此不被 Java 虚拟机认为是相同的。

了解了这一点之后，就可以理解代理模式的设计动机了。代理模式是为了保证 Java 核心库的类型安全。所有 Java 应用都至少需要引用 java.lang.Object 类，也就是说在运行的时候，java.lang.Object 这个类需要被加载到 Java 虚拟机中。如果这个加载过程由 Java 应用自己的类加载器来完成的话，很可能就存在多个版本的 java.lang.Object 类，而且这些类之间是不兼容的。通过代理模式，对于 Java 核心库的类的加载工作由引导类加载器来统一完成，保证了 Java 应用所使用的都是同一个版本的 Java 核心库的类，是互相兼容的。

不同的类加载器为相同名称的类创建了额外的名称空间。相同名称的类可以并存在 Java 虚拟机中，只需要用不同的类加载器来加载它们即可。不同类加载器加载的类之间是不兼容的，这就相当于在 Java 虚拟机内部创建了一个个相互隔离的 Java 类空间。这种技术在许多框架中都被用到，后面会详细介绍。

4．加载类的过程

在前面介绍类加载器的代理模式的时候，提到过类加载器会首先代理给其他类加载器来尝试加载某个类。这就意味着真正完成类的加载工作的类加载器和启动这个加载过程的类加载器，有可能不是同一个。真正完成类的加载工作是通过调用 defineClass 来实现的；而启动类的加载过程是通过调用 loadClass 来实现的。前者称为一个类的定义加载器

(Defining Loader)，后者称为初始加载器(Initiating Loader)。在 Java 虚拟机判断两个类是否相同的时候，使用的是类的定义加载器。也就是说，哪个类加载器启动类的加载过程并不重要，重要的是最终定义这个类的加载器。两种类加载器的关联之处在于：一个类的定义加载器是它引用的其他类的初始加载器。如类 com.example.Outer 引用了类 com.example.Inner，则由类 com.example.Outer 的定义加载器负责启动类 com.example.Inner 的加载过程。

方法 loadClass()抛出的是 java.lang.ClassNotFoundException 异常；方法 defineClass()抛出的是 java.lang.NoClassDefFoundError 异常。

类加载器在成功加载某个类之后，会把得到的 java.lang.Class 类的实例缓存起来。下次再请求加载该类的时候，类加载器会直接使用缓存的类的实例，而不会尝试再次加载。也就是说，对于一个类加载器实例来说，相同全名的类只加载一次，即 loadClass 方法不会被重复调用。

下面讨论另外一种类加载器：线程上下文类加载器。

5. 线程上下文类加载器

线程上下文类加载器(Context Class Loader)是从 JDK 1.2 开始引入的。类 java.lang.Thread 中的方法 getContextClassLoader()和 setContextClassLoader(ClassLoader cl)用来获取和设置线程的上下文类加载器。如果没有通过 setContextClassLoader(ClassLoader cl)方法进行设置的话，线程将继承其父线程的上下文类加载器。Java 应用运行的初始线程的上下文类加载器是系统类加载器。在线程中运行的代码可以通过此类加载器来加载类和资源。

前面提到的类加载器的代理模式并不能解决 Java 应用开发中会遇到的类加载器的全部问题。Java 提供了很多服务提供者接口(Service Provider Interface，SPI)，允许第三方为这些接口提供实现。常见的 SPI 有 JDBC、JCE、JNDI、JAXP 和 JBI 等。这些 SPI 的接口由 Java 核心库来提供，如 JAXP 的 SPI 接口定义包含在 javax.xml.parsers 包中。这些 SPI 的实现代码很可能是作为 Java 应用所依赖的 jar 包被包含进来，可以通过类路径(CLASSPATH)来找到，如实现了 JAXP SPI 的 Apache Xerces 所包含的 jar 包。SPI 接口中的代码经常需要加载具体的实现类。如 JAXP 中的 javax.xml.parsers.DocumentBuilderFactory 类中的 newInstance()方法用来生成一个新的 DocumentBuilderFactory 的实例。这里的实例的真正的类是继承自 javax.xml.parsers.DocumentBuilderFactory，由 SPI 的实现所提供的。如在 Apache Xerces 中，实现的类是 org.apache.xerces.jaxp.DocumentBuilderFactoryImpl。而问题在于，SPI 的接口是 Java 核心库的一部分，是由引导类加载器来加载的；SPI 实现的 Java 类一般是由系统类加载器来加载的。引导类加载器是无法找到 SPI 的实现类的，因为它只加载 Java 的核心库。它也不能代理给系统类加载器，因为它是系统类加载器的祖先类加载器。也就是说，类加载器的代理模式无法解决这个问题。

线程上下文类加载器正好解决了这个问题。如果不做任何的设置，Java 应用的线程的上下文类加载器默认就是系统上下文类加载器。在 SPI 接口的代码中使用线程上下文类加载器，就可以成功的加载到 SPI 实现的类。线程上下文类加载器在很多 SPI 的实现中都会用到。

下面介绍另外一种加载类的方法：Class.forName。

6．Class.forName

Class.forName 是一个静态方法，同样可以用来加载类。该方法有两种形式：Class.forName(String name、boolean initialize、ClassLoader loader)和 Class.forName(String className)。第一种形式的参数 name 表示的是类的全名；initialize 表示是否初始化类；loader 表示加载时使用的类加载器。第二种形式则相当于设置了参数 initialize 的值为 true，loader 的值为当前类的类加载器。Class.forName 的一个很常见的用法是在加载数据库驱动的时候。如 Class.forName("org.apache.derby.jdbc.EmbeddedDriver").newInstance()用来加载 Apache Derby 数据库的驱动。

11.3.2　JVM 启动时的三个类加载器

当 JVM(Java 虚拟机)启动时，会形成由三个类加载器组成的初始类加载器层次结构：

```
bootstrap classloader
|
extension classloader
|
system classloader
```

1．bootstrap classloader：引导(也称为原始)类加载器

负责加载 Java 的核心类。在 Sun 的 JVM 中，在执行 java 的命令中使用 -Xbootclasspath 选项或使用- D 选项指定 sun.boot.class.path 系统属性值可以指定附加的类。这个加载器的是非常特殊的，它实际上不是 java.lang.ClassLoader 的子类，而是由 JVM 自身实现的。大家可以通过执行以下代码来获得 bootstrap classloader 加载了那些核心类库：

```
URL[] urls=sun.misc.Launcher.getBootstrapClassPath().getURLs();
for (int i = 0; i < urls.length; i++) {
System.out.println(urls.toExternalForm());
}
```

在笔者计算机上的结果为：

```
file:/C:/j2sdk1.4.1 01/jre/lib/endorsed/dom.jar
file:/C:/j2sdk1.4.1 01/jre/lib/endorsed/sax.jar
file:/C:/j2sdk1.4.1 01/jre/lib/endorsed/xalan-2.3.1.jar
file:/C:/j2sdk1.4.1 01/jre/lib/endorsed/xercesImpl-2.0.0.jar
file:/C:/j2sdk1.4.1 01/jre/lib/endorsed/xml-apis.jar
file:/C:/j2sdk1.4.1 01/jre/lib/endorsed/xsltc.jar
file:/C:/j2sdk1.4.1 01/jre/lib/rt.jar
file:/C:/j2sdk1.4.1 01/jre/lib/i18n.jar
file:/C:/j2sdk1.4.1 01/jre/lib/sunrsasign.jar
file:/C:/j2sdk1.4.1 01/jre/lib/jsse.jar
file:/C:/j2sdk1.4.1 01/jre/lib/jce.jar
file:/C:/j2sdk1.4.1 01/jre/lib/charsets.jar
file:/C:/j2sdk1.4.1_01/jre/classes
```

这时大家知道了为什么我们不需要在系统属性 CLASSPATH 中指定这些类库了吧，因

为 JVM 在启动的时候就自动加载它们了。

2．extension classloader：扩展类加载器

负责加载 JRE 的扩展目录(JAVA_HOME/jre/lib/ext 或者由 java.ext.dirs 系统属性指定的)中 JAR 的类包。这为引入除 Java 核心类以外的新功能提供了一个标准机制。因为默认的扩展目录对所有从同一个 JRE 中启动的 JVM 都是通用的，所以放入这个目录的 JAR 类包对所有的 JVM 和 system classloader 都是可见的。在这个实例上调用方法 getParent()总是返回空值 null，因为引导加载器 bootstrap classloader 不是一个真正的 ClassLoader 实例。所以当大家执行以下代码时：

```
System.out.println(System.getProperty("java.ext.dirs"));
ClassLoader extensionClassloader=ClassLoader.getSystemClassLoader().getParent();
System.out.println("the parent of extension classloader :
"+extensionClassloader.getParent());
```

结果为：

```
C:/j2sdk1.4.1 01/jre/lib/ext
the parent of extension classloader : null
```

extension classloader 是 system classloader 的 parent，而 bootstrap classloader 是 extension classloader 的 parent，但它不是一个实际的 classloader，所以为 null。

3．system classloader：系统(也称为应用)类加载器

负责在 JVM 被启动时，加载来自在命令 java 中的-classpath 或者 java.class.path 系统属性或者 CLASSPATH 操作系统属性所指定的 JAR 类包和类路径。总能通过静态方法 ClassLoader.getSystemClassLoader()找到该类加载器。如果没有特别指定，则用户自定义的任何类加载器都将该类加载器作为它的父加载器。执行以下代码即可获得：

```
System.out.println(System.getProperty("java.class.path"));
```

输出结果则为用户在系统属性里面设置的 CLASSPATH。

classloader 加载类用的是全盘负责委托机制。所谓全盘负责，即是当一个 classloader 加载一个 Class 的时候，这个 Class 所依赖的和引用的所有 Class 也由这个 classloader 负责载入，除非是显式的使用另外一个 classloader 载入；委托机制则是先让 parent(父)类加载器(而不是 super，它与 parent classloader 类不是继承关系)寻找，只有在 parent 找不到的时候才从自己的类路径中去寻找。此外类加载还采用了 cache 机制，也就是如果 cache 中保存了这个 Class 就直接返回它，如果没有才从文件中读取和转换成 Class，并存入 cache，这就是为什么我们修改了 Class 但是必须重新启动 JVM 才能生效的原因。

每个 ClassLoader 加载 Class 的过程是：

(1) 检测此 Class 是否载入过(即在 cache 中是否有此 Class)，如果有到 8，如果没有到 2。

(2) 如果 parent classloader 不存在(没有 parent，那 parent 一定是 bootstrap classloader 了)，到 4。

(3) 请求 parent classloader 载入，如果成功到 8，不成功到 5。

(4) 请求 JVM 从 bootstrap classloader 中载入，如果成功到 8。

(5) 寻找 Class 文件(从与此 classloader 相关的类路径中寻找)，如果找不到则到 7。

(6) 从文件中载入 Class，到 8。

(7) 抛出 ClassNotFoundException 异常。

(8) 返回 Class。

其中(5)、(6)步我们可以通过覆盖 ClassLoader 的 findClass 方法来实现自己的载入策略。甚至覆盖 loadClass 方法来实现自己的载入过程。

类加载器的顺序是：先是 bootstrap classloader，然后是 extension classloader，最后才是 system classloader。大家会发现加载的 Class 越是重要的越在靠前面。这样做的原因是出于安全性的考虑，试想如果 system classloader“亲自”加载了一个具有破坏性的“java.lang.System”类的后果吧。这种委托机制保证了用户即使具有一个这样的类，也把它加入到了类路径中，但是它永远不会被载入，因为这个类总是由 bootstrap classloader 来加载的。大家可以执行一下以下的代码：

```
System.out.println(System.class.getClassLoader());
```

将会看到结果是 null，这就表明 java.lang.System 是由 bootstrap classloader 加载的，因为 bootstrap classloader 不是一个真正的 ClassLoader 实例，而是由 JVM 实现的，正如前面已经说过的。

下面就让我们来看看 JVM 是如何来为我们来建立类加载器的结构的。sun.misc.Launcher，顾名思义，当你执行 java 命令的时候，JVM 会先使用 bootstrap classloader 载入并初始化一个 Launcher，执行下面的代码：

```
System.out.println("the Launcher's classloader is
"+sun.misc.Launcher.getLauncher().getClass().getClassLoader());
```

结果为：

```
the Launcher's classloader is null
```

因为是用 bootstrap classloader 加载，所以 class loader 为 null。

Launcher 会根据系统和命令设定初始化好 class loader 结构，JVM 就用它来获得 extension classloader 和 system classloader,并载入所有的需要载入的 Class，最后执行 java 命令指定的带有静态的 main 方法的 Class。extension classloader 实际上是 sun.misc.Launcher$ExtClassLoader 类的一个实例，system classloader 实际上是 sun.misc.Launcher$AppClassLoader 类的一个实例，并且都是 java.net.URLClassLoader 的子类。

让我们来看看 Launcher 初始化过程的部分代码。

```
public class Launcher {
public Launcher() {
ExtClassLoader extclassloader;
try {
//初始化 extension classloader
extclassloader = ExtClassLoader.getExtClassLoader();
} catch(IOException ioexception) {
throw new InternalError("Could not create extension class loader");
```

```
}
try {
//初始化 system classloader，parent 是 extension classloader
loader = AppClassLoader.getAppClassLoader(extclassloader);
} catch(IOException ioexception1) {
throw new InternalError("Could not create application class loader");
}
//将 system classloader 设置成当前线程的 context classloader(将在后面加以介绍)
Thread.currentThread().setContextClassLoader(loader);
......
}
public ClassLoader getClassLoader() {
//返回 system classloader
return loader;
}
}
```

extension classloader 的部分代码：

```
static class Launcher$ExtClassLoader extends URLClassLoader {
public static Launcher$ExtClassLoader getExtClassLoader()
throws IOException
{
File afile[] = getExtDirs();
return (Launcher$ExtClassLoader)AccessController.doPrivileged(new
Launcher$1(afile));
}
private static File[] getExtDirs() {
//获得系统属性“java.ext.dirs”
String s = System.getProperty("java.ext.dirs");
File afile[];
if(s != null) {
StringTokenizer stringtokenizer = new StringTokenizer(s, File.pathSeparator);
int i = stringtokenizer.countTokens();
afile = new File;
for(int j = 0; j < i; j++)
afile[j] = new File(stringtokenizer.nextToken());
} else {
afile = new File[0];
}
return afile;
}
}
```

system classloader 的部分代码：

```
static class Launcher$AppClassLoader extends URLClassLoader
{
public static ClassLoader getAppClassLoader(ClassLoader classloader)
throws IOException
{
//获得系统属性“java.class.path”
String s = System.getProperty("java.class.path");
File afile[] = s != null ? Launcher.access$200(s) : new File[0];
return (Launcher$AppClassLoader)AccessController.doPrivileged(new
```

```
Launcher$2(s, afile, classloader));
}
```

由上述源代码可知，extension classloader 是使用系统属性“java.ext.dirs”设置类搜索路径的，并且没有 parent。system classloader 是使用系统属性“java.class.path”设置类搜索路径的，并且有一个 parent classloader。Launcher 初始化 extension classloader、system classloader，并将 system classloader 设置成为 context classloader，但是仅仅返回 system classloader 给 JVM。

这里怎么又出来一个 context classloader 呢？它有什么用呢？我们在建立一个线程 Thread 的时候，可以为这个线程通过 setContextClassLoader 方法来指定一个合适的 classloader 作为这个线程的 context classloader，当此线程运行的时候，我们可以通过 getContextClassLoader 方法来获得此 context classloader，就可以用它来载入我们所需要的 Class。默认的是 system classloader。利用这个特性，我们可以“打破”classloader 委托机制了，父 classloader 可以获得当前线程的 context classloader，而这个 context classloader 可以是它的子 classloader 或者其他的 classloader，那么父 classloader 就可以从其获得所需的 Class，这就打破了只能向父 classloader 请求的限制了。这个机制可以满足当我们的 classpath 是在运行时才确定，并由定制的 classloader 加载的时候，由 system classloader(即在 jvm classpath 中)加载的 class 可以通过 context classloader 获得定制的 classloader 并加载入特定的 class(通常是抽象类和接口，定制的 classloader 中是其实现)，例如 Web 应用中的 Servlet 就是用这种机制加载的。

现在我们了解了 classloader 的结构和工作原理，那么是如何实现在运行时的动态载入和更新呢？只要能够动态改变类搜索路径和清除 classloader 的 cache 中已经载入的 Class 即可，具体有如下两个方案可以实现。

(1) 继承一个 classloader，覆盖 loadclass 方法，动态的寻找 Class 文件并使用 defineClass 方法来。

(2) 只要重新使用一个新的类搜索路径来 new 一个 classloader 就行了，这样即更新了类搜索路径以便来载入新的 Class，也重新生成了一个空白的 cache(当然，类搜索路径不一定必须更改)。这个方案比较实用，我们几乎不用做什么工作，java.netURLClassLoader 正是一个符合要求的 classloader，我们可以直接使用或者继承它就可以了。

现在我们能够动态的载入 Class 了，这样我们就可以利用 newInstance 方法来获得一个 Object。但我们如何将此 Object 造型呢？可以将此 Object 造型成它本身的 Class 吗？首先让来分析一下 Java 源文件的编译和运行吧。javac 命令是调用 JAVA_HOME/lib/tools.jar 中的 com.sun.tools.javac.Main 的 compile 方法来编译：

```
public static int compile(String as[]);
public static int compile(String as[], PrintWriter printwriter);
```

返回 0 表示编译成功，字符串数组 as 则是我们用 javac 命令编译时的参数，以空格划分。例如：

```
javac -classpath c:/foo/bar.jar;. -d c:/c:/Some.java
```

字符串数组 as 为{"-classpath","c://foo//bar.jar;.","-d","c://","c://Some.java"}，如果带有

PrintWriter 参数，则会把编译信息出到这个指定的 PrintWriter 中。默认的输出是 System.err。

其中 Main 是由 JVM 使用 Launcher 初始化的 system classloader 载入的，根据全盘负责原则，编译器在解析这个 java 源文件时所发现的它所依赖和引用的所有 Class 也将由 system classloader 载入，如果 system classloader 不能载入某个 Class 时，编译器将抛出一个"cannot resolve symbol"错误。

所以首先编译就通不过，也就是编译器无法编译一个引用了不在 CLASSPATH 中的未知 Class 的 java 源文件，而由于拼写错误或者没有把所需类库放到 CLASSPATH 中，大家一定经常看到这个"cannot resolve symbol"这个编译错误。

其次，就是我们把这个 Class 放到编译路径中，成功地进行了编译，然后在运行的时候不把它放入到 CLASSPATH 中而利用我们自己的 classloader 来动态载入这个 Class，这时候也会出现"java.lang.NoClassDefFoundError"的违例，为什么呢？我们再来分析一下，首先调用这个造型语句的可执行的 Class 一定是由 JVM 使用 Launcher 初始化的 system classloader 载入的，根据全盘负责原则，当我们进行造型的时候，JVM 也会使用 system classloader 来尝试载入这个 Class 来对实例进行造型，自然在 system classloader 寻找不到这个 Class 时就会抛出"java.lang.NoClassDefFoundError"的违例。

现在让我们来总结一下，Java 文件的编译和 Class 的载入执行，都是使用 Launcher 初始化的 system classloader 作为类载入器的，我们无法动态的改变 system classloader，更无法让 JVM 使用我们自己的 classloader 来替换 system classloader，根据全盘负责原则，就限制了编译和运行时，我们无法直接显式的使用一个 system classloader 寻找不到的 Class，即我们只能使用 Java 核心类库，扩展类库和 CLASSPATH 中的类库中的 Class。

再尝试一下这种情况，我们把这个 Class 也放入到 CLASSPATH 中，让 system classloader 能够识别和载入。然后我们通过自己的 classloader 来从指定的 class 文件中载入这个 Class(不能够委托 parent 载入，因为这样会被 system classloader 从 CLASSPATH 中将其载入)，然后实例化一个 Object，并造型成这个 Class，这样 JVM 也识别这个 Class(因为 system classloader 能够定位和载入这个 Class 从 CLASSPATH 中)，载入的也不是 CLASSPATH 中的这个 Class，而是从 CLASSPATH 外动态载入的，这样总行了吧！十分不幸的是，这时会出现"java.lang.ClassCastException"违例，为什么呢？

我们也来分析一下，不错，我们虽然从 CLASSPATH 外使用我们自己的 classloader 动态载入了这个 Class，但将它的实例造型的时候是 JVM 会使用 system classloader 来再次载入这个 Class，并尝试将使用我们的自己的 classloader 载入的 Class 的一个实例造型为 system classloader 载入的这个 Class(另外的一个)。大家发现什么问题了吗？也就是我们尝试将从一个 classloader 载入的 Class 的一个实例造型为另外一个 classloader 载入的 Class，虽然这两个 Class 的名字一样，甚至是从同一个 class 文件中载入。但不幸的是 JVM 却认为这两个 Class 是不同的，即 JVM 认为不同的 classloader 载入的相同的名字的 Class(即使是从同一个 Class 文件中载入的)是不同的！这样做的原因我想大概也是主要出于安全性考虑，这样就保证所有的核心 Java 类都是 system classloader 载入的，我们无法用自己的 classloader 载入的相同名字的 Class 的实例来替换它们的实例。

看到这里，聪明的读者一定想到了该如何动态载入我们的 Class，实例化，造型并调用了吧！那就是利用面向对象的基本特性之一的多形性。我们把我们动态载入的 Class 的实例造型成它的一个 system classloader 所能识别的父类就行了！这是为什么呢？我们还是要再来分析一次。当我们用我们自己的 classloader 来动态载入这我们只要把这个 Class 的时候，发现它有一个父类 Class，在载入它之前 JVM 先会载入这个父类 Class，这个父类 Class 是 system classloader 所能识别的，根据委托机制，它将由 system classloader 载入，然后我们的 classloader 再载入这个 Class，创建一个实例，造型为这个父类 Class，注意了，造型成这个父类 Class 的时候(也就是上溯)是面向对象的 java 语言所允许的并且 JVM 也支持的，JVM 就使用 system classloader 再次载入这个父类 Class，然后将此实例造型为这个父类 Class。大家可以从这个过程发现这个父类 Class 都是由 system classloader 载入的，也就是同一个 class loader 载入的同一个 Class，所以造型的时候不会出现任何异常。而根据多形性，调用这个父类的方法时，真正执行的是这个 Class(非父类 Class)的覆盖了父类方法的方法。这些方法中也可以引用 system classloader 不能识别的 Class，因为根据全盘负责原则，只要载入这个 Class 的 classloader 即我们自己定义的 classloader 能够定位和载入这些 Class 就行了。

这样我们就可以事先定义好一组接口或者基类并放入 CLASSPATH 中，然后在执行的时候动态的载入实现或者继承了这些接口或基类的子类。还不明白吗？让我们来想一想 Servlet 吧，web application server 能够载入任何继承了 Servlet 的 Class 并正确的执行它们，不管它实际的 Class 是什么，就是都把它们实例化成为一个 Servlet Class，然后执行 Servlet 的 init、doPost、doGet 和 destroy 等方法的，而不管这个 Servlet 是从 web- inf/lib 下由 system classloader 的子 classloader(即定制的 classloader)动态载入，还是从 web-inf/classes 下由 system classloader 的子 classloader(即定制的 classloader)动态载入。综上所述，在 Applet 和 EJB 等容器中都是采用了这种机制。

classloader 虽然称为类加载器，但并不意味着只能用来加载 Class，我们还可以利用它也获得图片和音频文件等资源的 URL，当然这些资源必须在 CLASSPATH 中的 jar 类库中或目录下。我们来看 API 的 doc 中关于 ClassLoader 的两个寻找资源和 Class 的方法描述：

```
public URL getResource(String name)
```

可以通过上述方法指定的名字来查找资源，一个资源是一些能够被 Class 代码访问的在某种程度上依赖于代码位置的数据(图片、音频、文本等)。

一个资源的名字是以“/”号分隔确定资源的路径名的。这个方法将先请求 parent classloader 搜索资源，如果没有 parent，则会在内置在虚拟机中的 classloader(即 bootstrap classloader)路径中搜索。如果失败，这个方法将调用 findResource(String)来寻找资源。

```
public static URL getSystemResource(String name)
```

从用来载入类的搜索路径中查找一个指定名字的资源。这个方法使用 system class loader 来定位资源，即相当于 ClassLoader.getSystemClassLoader().getResource(name)。例如：

```
System.out.println(ClassLoader.getSystemResource("java/lang/String.class"));
```

的结果为：

```
jar:file:/C:/j2sdk1.6.23/jre/lib/rt.jar!/java/lang/String.class
```

表明 String.class 文件在 rt.jar 的 java/lang 目录中。

因此可以将图片等资源随同 Class 一同打包到 jar 类库中(当然，也可单独打包这些资源)并添加它们到 class loader 的搜索路径中，我们就无需关心这些资源的具体位置，让 class loader 来帮我们寻找了！

11.3.3 双亲委派模型

站在 Java 虚拟机的角度讲，只存在如下两种不同的类加载器：

(1) 启动类加载器(Bootstrap ClassLoader)，这个类加载器使用 C++语言实现，是虚拟机自身的一部分；

(2) 所有其他的类加载器，这些类加载器都由 Java 语言实现，独立于虚拟机外部，并且全都继承自抽象类 j ava.lang.ClassLoader。

从 Java 开发人员的角度来看，类加载器就还可以划分得更细致一些，绝大部分 Java 程序都会使用到以下三种系统提供的类加载器。

(1) 启动类加载器(Bootstrap ClassLoader)：前面已经介绍过，这个类加载器负责将存放在<JAVA_HOME>\lib 目录中的，或者被-Xbootclasspath 参数所指定的路径中的，并且是虚拟机识别的(仅按照文件名识别，如 rt.jar，名字不符合的类库即使放在 lib 目录中也不会被加载)类库加载到虚拟机内存中。启动类加载器无法被 Java 程序直接引用。

(2) 扩展类加载器(Extension ClassLoader)：这个加载器由 sun.misc.Launcher$ExtClass Loader 实现，它负责加载<JAVA_HOME>\lib\ext 目录中的，或者被 java.ext.dirs 系统变量所指定的路径中的所有类库，开发者可以直接使用扩展类加载器。

(3) 应用程序类加载器(Application ClassLoader)：这个类加载器由 sun.misc.Launcher$ AppClassLoader 来实现。由于这个类加载器是 ClassLoader 中的 getSystemClassLoader()方法的返回值，所以一般也称它为系统类加载器。它负责加载用户类路径(ClassPath)上所指定的类库，开发者可以直接使用这个类加载器，如果应用程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器。

我们的应用程序都是由这三种类加载器互相配合进行加载的，如果有必要，还可以加入自己定义的类加载器。

双亲委派模型要求除了顶层的启动类加载器外，其余的类加载器都应当有自己的父类加载器。这里类加载器之间的父子关系一般不会以继承(Inheritance)的关系来实现，而是都使用组合(Composition)关系来复用父加载器的代码。

类加载器的双亲委派模型在 JDK 1.2 期间被引入并被广泛应用于之后几乎所有的 Java 程序中，但它并不是一个强制性的约束模型，而是 Java 设计者们推荐给开发者们的一种类加载器实现方式。

双亲委派模型的工作过程是：如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成，每一个层次的类加载器

都是如此，因此所有的加载请求最终都应该传送到顶层的启动类加载器中，只有当父加载器反馈自己无法完成这个加载请求(它的搜索范围中没有找到所需的类)时，子加载器才会尝试自己去加载。

使用双亲委派模型来组织类加载器之间的关系，有一个显而易见的好处就是 Java 类随着它的类加载器一起具备了一种带有优先级的层次关系。例如类 java.lang.Object，它存放在 rt.jar 之中，无论哪一个类加载器要加载这个类，最终都是委派给启动类加载器进行加载，因此 Object 类在程序的各种类加载器环境中都是同一个类。相反，如果没有使用双亲委派模型，由各个类加载器自行去加载的话，如果用户自己写了一个名为 java.lang.Object 的类，并放在程序的 ClassPath 中，那系统中将会出现多个不同的 Object 类，Java 类型体系中最基础的行为也就无从保证，应用程序也将会变得一片混乱。如果您有兴趣的话，可以尝试去写一个与 rt.jar 类库中已有类重名的 Java 类，将会发现可以正常编译，但永远无法被加载运行。

双亲委派模型对于保证 Java 程序的稳定运作很重要，但它的实现却非常简单，实现双亲委派的代码都集中在 java.lang.ClassLoader 的 loadClass()方法之中，看看下面的演示代码。

```
protected synchronized Class<?> loadClass(String name, boolean resolve)
throws
   ClassNotFoundException
   {
   //首先，检查请求的类是否已经被加载过了
   Class c=findLoadedClass (name)j
   if(c==null){
   try{
   if (parent! null)  {
   c= parent.loadClass (name, false);
   } else if
   c= findBootstrapClassOrNull (name);
   }
   } catch (ClassNotFoundException e) {
   //如果父类加载器抛出 ClassNotFoundException
   //则说明父类加载器无法完成加载请求
   }
   if(c==null) {
   //在父类加载器无法加载的时候
   //再调用本身的 findClass 方法来进行类加载
   c= findClass (name);
   }
   }
   if (resolve) {
   resolveClass (c);
   }
   return c:
   }
```

11.3.4 破坏双亲委派模型

上文提到过双亲委派模型并不是一个强制性的约束模型，而是 Java 设计者们推荐给开

发者们的类加载器实现方式。在 Java 的世界里面大部分的类加载器都遵循这个模型，但也有例外的情况，到现在为止，双亲委派模型主要出现过三次较大规模的“被破坏”情况。

双亲委派模型的第一次“被破坏”其实发生在双亲委派模型出现之前——即 JDK 1.2 发布之前。由于双亲委派模型在 JDK 1.2 之后才被引入的，而类加载器和抽象类 java.lang.ClassLoader 则在 JDK 1.0 时代就已经存在，面对已经存在的用户自定义类加器的实现代码，Java 设计者们引入双亲委派模型时不得不做出一些妥协。为了向前兼容 JDK 1.2 之后的 java.lang.ClassLoader，添加了一个新的 protected 方法 findClass()，在此之前，用户去继承 java.lang.ClassLoader 的唯一目的就是为了重写 loadClass()方法，为虚拟机在进行类加载的时候会调用加载器的私有方法 loadClassInternal()，而这个方法的唯一逻辑就是去调用自己的 loadClass()。

上一节我们已经看过 loadClass()方法的代码，双亲委派的具体逻辑就实现在这个方法之中，JDK 1.2 之后已不提倡用户再去覆盖 loadClass()方法，而应当把自己的类加载逻辑写到 findClass()方法中，在 loadClass()方法的逻辑里如果父类加载失败，则会调用自己的 findClass()方法来完成加载，这样就可以保证新写出来的类加载器是符合双亲委派规则的。

双亲委派模型的第二次“被破坏”是由这个模型自身的缺陷所导致的，双亲委派很好地解决了各个类加载器的基础类的统一问题(越基础的类由越上层的加载器进行加载)，基础类之所以被称为“基础”，是因为它们总是作为被用户代码调用的 API，但世事往往没有绝对的完美，如果基础类又要调用回用户的代码，那该怎么办？

这并非是不可能的事情，一个典型的例子便是 JNDI 服务，JNDI 现在已经是 Java 的标准服务，它的代码由启动类加载器去加载(在 JDK 1.3 时代放进去的 rt.jar)，但 JNDI 的目的就是对资源进行集中管理和查找，它需要调用由独立厂商实现并部署在应用程序的 ClassPath 下的 JNDI 接口提供者(Service Provider Interface，SPI)的代码，但启动类加载器不可能“认识”这些代码啊！那该怎么办？

为了解决这个困境，Java 设计团队只好引入了一个不太优雅的设计：线程上下文类加载器(Thread Context ClassLoader)。这个类加载器可以通过 java.lang.Thread 类的 setContextClassLoaser()方法进行设置，如果创建线程时还未设置，它将会从父线程中继承一个；如果在应用程序的全局范围内都没有设置过，那么这个类加载器默认就是应用程序类加载器。

有了线程上下文类加载器，就可以做一些“舞弊”的事情了，JNDI 服务使用这个线程上下文类加载器去加载所需要的 SPI 代码，也就是父类加载器请求子类加载器去完成类加载的动作，这种行为实际上就是打通了双亲委派模型的层次结构来逆向使用类加载器，已经违背了双亲委派模型的一般性原则，但这也是无可奈何的事情。Java 中所有涉及 SPI 的加载动作基本上都采用这种方式，例如 JNDI、JDBC、JCE、JAXB 和 JBI 等。

双亲委派模型的第三次“被破坏”是由于用户对程序动态性的追求而导致的，这里所说的“动态性”指的是当前一些非常“热”门的名词：代码热替换(HotSwap)、模块热部署(Hot Deployment)等，说白了就是希望应用程序能像我们的电脑外设那样，插上鼠标或 U 盘，不用重启机器就能立即使用，鼠标有问题或要升级就换个鼠标，不用停机也不用重

启。对于个人电脑来说，重启一次其实没有什么大不了的，但对于一些生产系统来说，关机重启一次可能就要被列为生产事故，这种情况下热部署就对软件开发者，尤其是企业级软件开发者具有很大的吸引力。

在 JSR-297、JSR-277 规范从纸上标准变成真正可运行的程序之前，OSGi 是当前业界“事实上”的 Java 模块化标准，而 OSGi 实现模块化热部署的关键则是它自定义的类加载器机制的实现。每一个程序模块(OSGi 中称为 Bundle)都有一个自己的类加载器，当需要更换一个 Bundle 时，就把 Bundle 连同类加载器一起换掉以实现代码的热替换。

在 OSGi 环境下，类加载器不再是双亲委派模型中的树状结构，而是进一步发展为网状结构，当收到类加载请求时，OSGi 将按照下面的顺序进行类搜索：

(1) 将以 java.*开头的类，委派给父类加载器加载。

(2) 否则，将委派列表名单内的类，委派给父类加载器加载。

(3) 否则，将 Import 列表中的类，委派给 Export 这个类的 Bundle 的类加载器加载。

(4) 否则，查找当前 Bundle 的 ClassPath，使用自己的类加载器加载。

(5) 否则，查找类是否在自己的 Fragment Bundle 中，如果在，则委派给 Fragment Bundle 的类加载器加载。

(6) 否则，查找 Dynamic Import 列表的 Bundle，委派给对应 Bundle 的类加载器加载。

(7) 否则，类查找失败。

上面的查找顺序中只有开头两点仍然符合双亲委派规则，其余的类查找都是在平级的类加载器中进行的。

笔者虽然使用了“被破坏”这个词来形容上述不符合双亲委派模型原则的行为，这里“被破坏”并不带 OSGi 的感情色彩。只要有足够意义和理由，突破 Java 的原则就可以算作一种创新。正如 OSGi 中的类加载器并不符合传统的双亲委派的类加载器，并且业界对其为了实现热部署而带来的额外的高复杂度还存在不少争议，但在 Java 程序员中基本有一个共识：OSGi 中对类加载器的使用是很值得学习的，弄懂了 OSGi 的实现，自然就明白了类加载器的精粹。

11.3.5　开发自己的类加载器

虽然在绝大多数情况下，系统默认提供的类加载器实现已经可以满足需求。但是在某些情况下，您还是需要为应用开发出自己的类加载器。比如您的应用通过网络来传输 Java 类的字节代码，为了保证安全性，这些字节代码经过了加密处理。这个时候您就需要自己的类加载器来从某个网络地址上读取加密后的字节代码，接着进行解密和验证，最后定义出要在 Java 虚拟机中运行的类来。下面将通过两个具体的实例来说明类加载器的开发。

1) 文件系统类加载器

第一个类加载器用来加载存储在文件系统上的 Java 字节代码，完整的实现代码如下。

```
public class FileSystemClassLoader extends ClassLoader {
    private String rootDir;
    public FileSystemClassLoader(String rootDir) {
        this.rootDir = rootDir;
    }
```

```
    protected Class<?> findClass(String name) throws
ClassNotFoundException {
        byte[] classData = getClassData(name);
        if (classData == null) {
            throw new ClassNotFoundException();
        }
        else {
            return defineClass(name, classData, 0, classData.length);
        }
    }
    private byte[] getClassData(String className) {
        String path = classNameToPath(className);
        try {
            InputStream ins = new FileInputStream(path);
            ByteArrayOutputStream baos = new ByteArrayOutputStream();
            int bufferSize = 4096;
            byte[] buffer = new byte[bufferSize];
            int bytesNumRead = 0;
            while ((bytesNumRead = ins.read(buffer)) != -1) {
                baos.write(buffer, 0, bytesNumRead);
            }
            return baos.toByteArray();
        } catch (IOException e) {
            e.printStackTrace();
        }
        return null;
    }
    private String classNameToPath(String className) {
        return rootDir + File.separatorChar
                + className.replace('.', File.separatorChar) + ".class";
    }
}
```

在上述代码中，类 FileSystemClassLoader 继承自类 java.lang.ClassLoader。在表 11-1 中列出的 java.lang.ClassLoader 类的常用方法中，一般来说，自己开发的类加载器只需要覆写 findClass(String name)方法即可。java.lang.ClassLoader 类的方法 loadClass()封装了前面提到的代理模式的实现。该方法会首先调用 findLoadedClass()方法来检查该类是否已经被加载过；如果没有加载过的话，会调用父类加载器的 loadClass()方法来尝试加载该类；如果父类加载器无法加载该类的话，就调用 findClass()方法来查找该类。因此，为了保证类加载器都正确实现代理模式，在开发自己的类加载器时，最好不要覆写 loadClass()方法，而是覆写 findClass()方法。

类 FileSystemClassLoade 的 findClass()方法首先根据类的全名在硬盘上查找类的字节代码文件(.class 文件)，然后读取该文件内容，最后通过 defineClass()方法来把这些字节代码转换成 java.lang.Class 类的实例。

2) 网络类加载器

下面将通过一个网络类加载器来说明如何通过类加载器来实现组件的动态更新。即基本的场景是：Java 字节代码(.class)文件存放在服务器上，客户端通过网络的方式获取字节代码并执行。当有版本更新的时候，只需要替换掉服务器上保存的文件即可。通过类加载器可以比较简单的实现这种需求。

类 NetworkClassLoader 负责通过网络下载 Java 类字节代码并定义出 Java 类。它的实现与 FileSystemClassLoader 类似。在通过 NetworkClassLoader 加载了某个版本的类之后，一般有两种做法来使用它。第一种做法是使用 Java 反射 API。另外一种做法是使用接口。需要注意的是，并不能直接在客户端代码中引用从服务器上下载的类，因为客户端代码的类加载器找不到这些类。使用 Java 反射 API 可以直接调用 Java 类的方法。而使用接口的做法则是把接口的类放在客户端中，从服务器上加载实现此接口的不同版本的类。在客户端通过相同的接口来使用这些实现类。网络类加载器的具体代码保存在本书售后技术持的群共享中，欢迎读者下载使用。

11.3.6　类加载器与 Web 容器

对于运行在 Java EE™ 容器中的 Web 应用来说，类加载器的实现方式与一般的 Java 应用有所不同。不同的 Web 容器的实现方式也会有所不同。以 Apache Tomcat 来说，每个 Web 应用都有一个对应的类加载器实例。该类加载器也使用代理模式，所不同的是它是首先尝试去加载某个类，如果找不到再代理给父类加载器。这与一般类加载器的顺序是相反的。这是 Java Servlet 规范中的推荐做法，其目的是使得 Web 应用自己的类的优先级高于 Web 容器提供的类。这种代理模式的一个例外是：Java 核心库的类是不在查找范围之内的。这也是为了保证 Java 核心库的类型安全。

绝大多数情况下，Web 应用的开发人员不需要考虑与类加载器相关的细节。下面给出几条简单的原则：

- 每个 Web 应用自己的 Java 类文件和使用的库的 jar 包，分别放在 WEB-INF/classes 和 WEB-INF/lib 目录下面。
- 多个应用共享的 Java 类文件和 jar 包，分别放在 Web 容器指定的由所有 Web 应用共享的目录下面。
- 当出现找不到类的错误时，检查当前类的类加载器和当前线程的上下文类加载器是否正确。

介绍完类加载器与 Web 容器的关系之后，下面介绍它与 OSGi 的关系。

11.3.7　类加载器与 OSGi

OSGi 是 Java 上的动态模块系统。它为开发人员提供了面向服务和基于组件的运行环境，并提供标准的方式用来管理软件的生命周期。OSGi 已经被实现和部署在很多产品上，在开源社区也得到了广泛的支持。Eclipse 就是基于 OSGi 技术来构建的。

OSGi 中的每个模块(Bundle)都包含 Java 包和类。模块可以声明它所依赖的需要导入(Import)的其他模块的 Java 包和类(通过 Import-Package)，也可以声明导出(Export)自己的包和类，供其他模块使用(通过 Export-Package)。也就是说需要能够隐藏和共享一个模块中的某些 Java 包和类。这是通过 OSGi 特有的类加载器机制来实现的。OSGi 中的每个模块都有对应的一个类加载器。它负责加载模块自己包含的 Java 包和类。当它需要加载 Java 核心库的类时(以 java 开头的包和类)，它会代理给父类加载器(通常是启动类加载器)来完成。当它需要加载所导入的 Java 类时，它会代理给导出此 Java 类的模块来完成加

载。模块也可以显式的声明某些 Java 包和类，必须由父类加载器来加载。只需要设置系统属性 org.osgi.framework.bootdelegation 的值即可。

假设有两个模块 bundleA 和 bundleB，它们都有自己对应的类加载器 classLoaderA 和 classLoaderB。在 bundleA 中包含类 com.bundleA.Sample，并且该类被声明为导出的，也就是说可以被其他模块所使用的。bundleB 声明了导入 bundleA 提供的类 com.bundleA.Sample，并包含一个类 com.bundleB.NewSample 继承自 com.bundleA.Sample。在 bundleB 启动的时候，其类加载器 classLoaderB 需要加载类 com.bundleB.NewSample，进而需要加载类 com.bundleA.Sample。由于 bundleB 声明了类 com.bundleA.Sample 是导入的，classLoaderB 把加载类 com.bundleA.Sample 的工作代理给导出该类的 bundleA 的类加载器 classLoaderA。classLoaderA 在其模块内部查找类 com.bundleA.Sample 并定义它，所得到的类 com.bundleA.Sample 实例就可以被所有声明导入了此类的模块使用。对于以 java 开头的类，都是由父类加载器来加载的。如果声明了系统属性 org.osgi.framework.bootdelegation=com.example.core.*，那么对于包 com.example.core 中的类，都是由父类加载器来完成的。

OSGi 模块的这种类加载器结构，使得一个类的不同版本可以共存在 Java 虚拟机中，带来了很大的灵活性。不过它的这种不同，也会给开发人员带来一些麻烦，尤其当模块需要使用第三方提供的库的时候。下面提供几条比较好的建议：

- 如果一个类库只有一个模块使用，把该类库的 jar 包放在模块中，在 Bundle-ClassPath 中指明即可。
- 如果一个类库被多个模块共用，可以为这个类库单独的创建一个模块，把其他模块需要用到的 Java 包声明为导出的。其他模块声明导入这些类。
- 如果类库提供了 SPI 接口，并且利用线程上下文类加载器来加载 SPI 实现的 Java 类，有可能会找不到 Java 类。如果出现了 NoClassDefFoundError 异常，首先检查当前线程的上下文类加载器是否正确。通过 Thread.currentThread().getContextClassLoader()就可以得到该类加载器。该类加载器应该是该模块对应的类加载器。如果不是的话，可以首先通过 class.getClassLoader()来得到模块对应的类加载器，再通过 Thread.currentThread().setContextClassLoader()来设置当前线程的上下文类加载器。

Java

第 12 章

研究高效之魂

执行引擎是 Java 虚拟机最核心的组成部分之一。“虚拟机”是一个相对于“物理机”的概念，这两种机器都有代码执行能力，其区别是物理机的执行引擎是直接建立在处理器、硬件、指令集和操作系统层面上的，而虚拟机的执行引擎则是由自己实现的。本章将讲解从栈帧、方法调用和字节码解释执行引擎入手，详细讲解 Java 虚拟机实现高效处理的基本知识。

12.1 虚拟机的字节码

Java 虚拟机一般是用软件模拟出来的。所以它的工作机制相对硬件机器有如下三点不同。

(1) 它没有寄存器，而是用操作数栈(Operand Stack)来代替，使用起来更加简单。

(2) 它使用精简指令集。它的指令集相对容易掌握。

(3) 它有一些指令是为面向对象而设计的，比如它有一个指令用来取出指定的对象实例的某个字段。

在 Java 虚拟机规范中制定了虚拟机字节码执行引擎的概念模型，这个概念模型成为各种虚拟机执行引擎的统一外观(Facade)。在不同的虚拟机实现里面，执行引擎在执行 Java 代码的时候可能有解释执行(通过解释器执行)和编译执行(通过即时编译器产生本地代码执行)两种选择，也可能两者兼备，甚至还可能包含几个不同级别的编译器执行引擎。但从外观上看起来，所有的 Java 虚拟机的执行引擎都是一致的：输入的是字节码文件，处理过程是字节码解析的等效过程，输出的是执行结果。本章将主要从概念模型的角度来讲解虚拟机的方法调用和字节码执行。

执行引擎在执行 Java 代码的时候，可以选择解释执行(通过解释器执行)和编译执行(通过即使编译器产生本地代码执行)两种选择。

栈帧(Stack Frame)是用于支持虚拟机进行方法调用和方法执行的数据结构，它是虚拟机运行时数据区中的虚拟机栈(Virtual Machine Stack)的栈元素。栈帧存储了方法的局部变量表，操作数栈，动态连接和方法返回地址等信息。每一个方法调用的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程。

一个线程中的方法调用链可能很长，很多方法都同时处于执行状态，对于执行引擎来说，活动线程中，只有栈顶的栈帧是有效的，成为 Current Stack Frame。这个栈帧所关联的方法称为当前方法(Current Method)。执行引擎所运行的所有字节码指令都只针对当前栈帧进行操作。

在虚拟机运行时，在其内存结构中的栈内有很多帧。每个帧对应一次方法执行。在帧里有两个重要的数据结构：操作数栈和局部变量表。

1) 操作数栈

操作数栈是用来存放指令的参数的。比如我们做 C=A+B，先把 A 放进操作数栈，再把 B 放进去，然后执行一个加法指令。栈里的 A 和 B 会被清除，A+B 的结果被放进栈里。再来一个指令，把栈顶的数据放到 C 变量里。

我们知道在 X86 系统中有很多寄存器，例如 EAX、EBX、ECX、EDX、ESP 等等，要掌握每个寄存器的用途还真需要一些时间。而 Java 虚拟机就简单多了，在虚拟机中你看不到那么多寄存器概念了，如果说有，那就是只有一个的程序指针寄存器，用来指定程序执行到的代码位置。

2) 局部变量表

局部变量表里存放局部变量。局部变量的个数是在编译时就确定的，所以局部变量表的大小对每个方法来说是确定的。局部变量表里的变量用下标 0、1、2、3 等来引用。

下面我们借个例子来看看一个方法是如何被执行的。假如存在下面一个方法：

```
void method(int a,int b){
int c = a + b;
}
```

在方法调用时，局部变量表里已经放了如下三个变量：

- 0：this 因为这个方法不是 static 的，它属于某个实例，所以有 this 这个隐含的局部变量。
- 1：a。
- 2：b。

局部变量表的大小是 4，下标 3 的位置是给变量 c 使用的。上边的 Java 代码会被编译成字节码，虚拟机会进行如下执行过程：

```
iload 0 //把局部变量整数 a 放进操作数栈
iload 1 //把局部变量整数 b 放进操作数栈
iadd //把栈顶的两个整数加起来，同时把它们从栈里清除掉，把结果放到栈顶
istore_3 //把栈顶的整数放到局部变量表的下标 3 的位置
```

javap 命令可以用来反汇编 java class 文件，可以看到字节码和局部变量表等。在没有源代码的情况下，使用 javap 读读字节码也能解决一些问题。

除了上述两种重要的数据结构外，还有如下两个重要的概念。

1) 动态连接

每个栈帧都包含一个指向运行时常量池中该栈帧所属方法的引用，持有这个引用是为了支持方法调用过程中的动态连接。

字节码中的方法调用指令就以常量池中指向方法的符号引用为参数。这些符号引用一部分会在类加载阶段或第一次使用的时候转换为直接引用，成为静态解析，另一部分会在每一次运行时转换为直接引用，成为动态连接。

2) 方法返回地址

有如下两种方式退出当前执行的方法：

- 执行引擎遇到任意一个方法返回的字节码指令，这种方法称为正常完成出口；
- 在方法执行过程中遇到无法处理的异常，这种方法称为异常完成出口。

无论哪一种方法，当方法退出后，都需要返回到调用者的位置。当正常退出时，调用者的 PC 计数器值可以作为返回地址，栈帧中很可能会保存这个计数器值，而异常退出时，返回地址要通过异常处理器表来确定。

方法退出的过程实际上是将当前栈帧出战，并恢复上层方法的局部变量表和操作数栈，把返回值压入调用者的操作数栈中。

12.2　栈帧的结构

在本节将要讲解的是 JVM 运行时栈帧的结构，并详细讲解操作数栈和局部变量表的基本知识，为学习本书后面的知识打下基础。

12.2.1　什么是栈帧

每当启用一个线程时，JVM 就为他分配一个 Java 栈，栈是以帧为单位保存当前线程的运行状态。某个线程正在执行的方法称为当前方法，当前方法使用的栈帧称为当前帧，当前方法所属的类称为当前类，当前类的常量池称为当前常量池。当线程执行一个方法时，它会跟踪当前常量池。

每当线程调用一个 Java 方法时，JVM 就会在该线程对应的栈中压入一个帧，这个帧自然就成了当前帧。当执行这个方法时，它使用这个帧来存储参数、局部变量、中间运算结果等。

Java 栈上的所有数据都是私有的。任何线程都不能访问另一个线程的栈数据。所以我们不用考虑多线程情况下栈数据访问同步的情况。

像方法区和堆一样，Java 栈和帧在内存中也不必是连续的，帧可以分布在连续的栈里，也可以分布在堆里。

栈帧(Stack Frame)是用于支持虚拟机进行方法调用和方法执行的数据结构，是虚拟机运行时数据区中的虚拟机栈(Virtual Machine Stack)的栈元素。栈帧存储了方法的局部变量表、操作数栈、动态连接和方法返回地址等信息。每一个方法从调用开始到执行完成的过程，就对应着一个栈帧在虚拟机栈里面从入栈到出栈的过程。

每一个栈帧都包括了局部变量表、操作数栈、动态连接、方法返回地址和一些额外的附加信息。在编译程序代码的时候，栈帧中需要多大的局部变量表、多深的操作数栈都已经完全确定了，并且写入到方法表的 Code 属性中的标记@后面，因此一个栈帧需要分配多少内存，不会受到程序运行期变量数据的影响，而仅仅取决于具体的虚拟机实现。

一个线程中的方法调用链可能会很长，很多方法都同时处于执行状态。对于执行引擎来讲，活动线程中，只有栈顶的栈帧是有效的，称为当前栈帧(Current Stack Frame)，这个栈帧所关联的方法称为当前方法(Current Method)。执行引擎所运行的所有字节码指令都只针对当前栈帧进行操作，栈帧的概念结构如图 12-1 所示。



图 12-1　栈帧的概念结构

下面详细讲解栈帧中的局部变量表、操作数栈、动态连接、方法返回地址等各个部分的作用和数据结构。

12.2.2　局部变量表

1) 局部变量

局部变量(Local Variable)是指作用域和生命周期都局限在所在函数或过程范围内的变量，它是相对于全局变量(Global Variable)而言的。编译器在为局部变量分配空间时通常有两种做法：使用寄存器和使用栈。

寄存器的访问速度快，但数量和空间有限，所以像字符串或数组不适合分配在寄存器中。编译器通常只会把频繁使用的临时变量分配在寄存器中，比如 for 循环中的循环变量。当编译器的优化选项打开时，编译器会充分利用可用的寄存器来给临时变量使用，以提高程序的性能。对于调试版本，优化选项默认是关闭的，编译器会在栈上分配所有的变量。在 C/C++程序中，可以在声明变量时加上 register 关键字，请求编译器在可能的情况下将该变量分配在寄存器中，但不能保证所描述的变量一定被分配在就存寄存器中。大多数时候，编译器还是根据全局设置和编译器自身的逻辑来决定是否把一个变量分配在寄存器中。

编译器会在编译阶段根据变量的特征和优化选项为每个局部变量选择以上两种分配方法之一。大多数的局部变量都是分配在栈上的。栈上的变量会随着函数的调用和返回而自动分配和释放，所以栈有时也称为自动内存。

局部变量的分配和释放是由编译器插入的代码通过调整栈指针(Stack Pointer)的位置来完成。编译器在编译时，会计算当前的代码块中所声明的所有局部变量所需要的空间，并将其按照内存对齐要求的最接近整数值。在 32 位系统中，内存分配时按 4 字节对齐的，这意味着不满 4 字节的空间会按 4 字节来分配。

计算好所需的空间后，编译器会插入适当的指令来调整栈指针 ESP，为局部变量分配空间，有两种方式调整 ESP 的值，一种是直接进行加减运算，另一种是 PUSH 和 POP 指令。

当我们看到反汇编代码时，常见到的指令是 esp+n 这种相对的地址，用 esp 进行标注的缺点是 ESP 的值是不稳定的，当 ESP 的值变化了，引用变量的偏移值也要变化。为了解决以上问题，x86 CPU 设计了另一个寄存器，这就是 EBP 寄存器。EBP 的全称是 Extended Base Pointer，即扩展的基地址指针。使用 EBP 寄存器，函数可以把自己将要使用的栈空间的基准地址记录下来，然后使用这个基地址来引用局部变量和参数。在同一函数内，EBP 寄存器的值是保持不变的，这样函数的局部变量就有了一个固定的参照物。

通常，一个函数在入口处将当时的 EBP 值压入堆栈，然后把 ESP 值(栈顶)赋值给 EBP，这样 EBP 中的地址就是进入本函数时的栈顶地址。因此在 EBP 地址的上面便是这个函数使用的栈空间，它下面(地址值递增方向)是父函数使用的空间例如 EBP+4 指向的是 CALL 指令压入函数的返回地址。EBP+8 是父函数压在栈上的第一个参数，EBP+0xC 是第二个参数。依次类推，EBP-n 是第一个局部变量的起始地址(n 为变量的长度)。

因为在将栈顶地址(ESP)赋给 EBP 寄存器之前先把旧的 EBP 值保存在栈中，所以 EBP

寄存器所指向的栈单元中保存的是前一个 EBP 寄存器的值，这通常也就是父函数的 EBP 值。类似的父函数的 EBP 所指向的栈单元中保存的是更上一层函数的 EBP 值，依此类推，直到当前线程的最顶层函数。这也正是栈回溯的基本原理。

2) 局部变量表

局部变量表是一组变量值存储空间，用于存放方法参数和方法内部定义的局部变量。在 Java 程序被编译为 Class 文件时，就在方法的 Code 属性的 max locals 数据项中确定了该方法所需要分配的最大局部变量表的容量。

局部变量表的容量以变量槽(Variable Slot，下称 Slot)为最小单位，虚拟机规范中并没有明确指明一个 Slot 应占用的内存空间大小，只是很有“导向性”地说明每个 Slot 都应该能存放一个 boolean、byte、char、short、mt、float、reference 或 returnAddress 类型的数据，这种描述与明确指出“每个 Slot 占用 32 位长度的内存空间”有一些差别，它允许 Slot 的长度随着处理器、操作系统或虚拟机的不同而发生变化。不过无论如何，即使在 64 位虚拟机中使用了 64 位长度的内存空间来实现一个 Slot，虚拟机仍要使用对齐和补白的手段让 Slot 在外观上看起来与 32 位虚拟机中的一致。

既然前面提到了数据类型，在此顺便说一下，一个 Slot 可以存放一个 32 位以内的数据类型，Java 中占用 32 位以内的数据类型有 boolean、byte、char、short、int、float、reference 和 returnAddress 八种类型。前面六种不需要多加解释，大家都认识，而后面的 reference 是对象的引用。虚拟机规范既没有说明它的长度，也没有明确指出这个引用应有怎样的结构，但是一般来说，虚拟机实现至少都应当能从此引用中直接或间接地查找到对象在 Java 堆中的起始地址索引和方法区中的对象类型数据。而 returnAddress 是为字节码指令 jsr、jsr_w 和 ret 服务的，它指向了一条字节码指令的地址。

对于 64 位的数据类型，虚拟机会以高位在前的方式为其分配两个连续的 Slot 空间。

Java 语言中明确规定的 64 位的数据类型只有 long 和 double 两种(reference 类型则可能是 32 位也可能是 64 位)。值得一提的是，这里把 long 和 double 数据类型分割存储的做法与“long 和 double 的非原子性协定”中把一次 long 和 double 数据类型读写分割为两次 32 位读写的做法类似，读者阅读到 Java 内存模型时可以对比一下。不过，由于局部变量表建立在线程的堆栈上，是线程私有的数据，无论读写两个连续的 Slot 是否是原子操作，都不会引起数据安全问题。

虚拟机通过索引定位的方式使用局部变量表，索引值的范围是从 0 开始到局部变量表最大的 Slot 数量。如果是 32 位数据类型的变量，索引，z 就代表了使用第 z 个 Slot，如果是 64 位数据类型的变量，则说明要使用第 z 和第 n+l 两个 Slot。

在方法执行时，虚拟机是使用局部变量表完成参数值到参数变量列表的传递过程的，如果是实例方法(非 static 的方法)，那么局部变量表中第 0 位索引的 Slot 默认是用于传递方法所属对象实例的引用，在方法中可以通过关键字“this”来访问这个隐含的参数。其余参数则按照参数表的顺序来排列，占用从 1 开始的局部变量 Slot，参数表分配完毕后，再根据方法体内部定义的变量顺序和作用域分配其余的 Slot。

局部变量表中的 Slot 是可重用的，方法体中定义的变量，其作用域并不一定会覆盖整个方法体，如果当前字节码 PC 计数器的值已经超出了某个变量的作用域，那么这个变量

对应的 Slot 就可以交给其他变量使用。这样的设计不仅仅是为了节省栈空间，在某些情况下 Slot 的复用会直接影响到系统的垃圾收集行为，请看下面的演示代码。

```
public static void main(String[] args) () {
        bytefl placeholder = new byte[64 * 1024 * 1024] ;
           System.gc () ;
        }
```

上述演示代码代码非常简单，功能是向内存填充了 64MB 的数据，然后通知虚拟机进行垃圾收集。我们在虚拟机运行参数中加上“-verbose:gc”来看看垃圾收集的过程，发现在 System.gc()运行后并没有回收这 64MB 的内存，下面是运行的结果：

```
[GC 66846K->65824K(125632K),  0.0032678 secs]
[Full GC 65824K->65746K(125632K),  0.0064131 secs]
```

没有回收 placeholder 所占的内存能说得过去，因为在执行 System.gc()时，变量 placeholder 还处于作用域之内，虚拟机自然不敢回收 placeholder 的内存。我们把代码修改一下，变成如下演示代码。

```
public static void main(String[] args) () {
         {
        byte[] placeholder = new byte[64 * 1024 * 1024] ;
         }
        System.gc ( ) ;
  }
```

加入了花括号之后，placeholder 的作用域被限制在花括号之内，从代码逻辑上讲，在执行 System.gc()的时候，placeholder 已经不可能再被访问了，但执行一下这段程序，会发现运行结果如下，还是有 64MB 的内存没有被回收，这又是为什么呢？

```
[GC 66846K->65888K(125632K),0.0009397 secs]
[Full  GC  65888K->65746:K (125632K),  0.0051574  secs]
```

在解释为什么之前，我们先对这段代码进行第二次修改，在调用 System.gc()之前加入一行代码“int a=0;”，变成如下演示代码。

```
public static void main(String[] args) () {
         {
        byte[] placeholder = new byte[64 * 1024 * 1024] ;
         }
         int a= 0;
        System.gc ( ) ;
  }
```

这个修改看起来很莫名其妙，但运行 一下程序，却发现这次内存真的被正确回收了：

```
[GC  664 01K- >65778K (12 5632K),  0.00354 71   secs]
[Full GC 65778K->2181 (125632K),0.0140596 secsJ
```

在上述三段演示代码中，placeholder 能否被回收的根本原因就是：局部变量表中的 Slot 是否还存有关于 placeholder 数组对象的引用。在第一次修改中，代码虽然已经离开了 placeholder 的作用域，但在此之后，没有任何对局部变量表的读写操作，placeholder 原本

所占用的 Slot 还没有被其他变量所复用，所以作为 GC Roots 一部分的局部变量表仍然保持着对它的关联。这种关联没有被及时打断，在绝大部分情况下影响都很轻微。但如果遇到一个方法，其后面的代码有一些耗时很长的操作，而前面又定义了占用了大量内存、实际上已经不会再被使用的变量，手动将其设置为 null 值(用来代替“int a 0;”，把变量对应的局部变量表 Slot 清空)就不是一个毫无意义的操作，这种操作可以作为一种在极特殊情形(对象占用内存大、此方法的栈帧长时间不能被回收、方法调用次数达不到 JIT 的编译条件)下的“奇技”来使用。但不应当对赋 null 值操作有过多的依赖，也没有必要把它当做一个普遍的编码方法来推广。以恰当的变量作用域来控制变量回收时间才是最优雅的解决方法，如上述第三段代码那样的场景并不多见。

另外，赋 null 值的操作在经过虚拟机 JIT 编译器优化之后会被消除掉，这时候将变量设置为 null 实际上是没有意义的。字节码被编译为本地代码后，对 GC Roots 的枚举也与解释执行时期有所差别。上述第二段代码在经过 JIT 编译后，System.gc()执行时就可以正确地回收掉内存，而无需写成上述第三段代码的样子。

关于局部变量表，还有一点可能会对实际开发产生影响，就是局部变量不像前面介绍的类变量那样存在“准备阶段”。通过前一章的讲解，我们已经知道类变量有两次赋初始值的过程，一次在准备阶段，赋予系统初始值；另外一次在初始化阶段，赋予程序员定义的初始值。因此即使在初始化阶段程序员没有为类变量赋值也没有关系，类变量仍然具有一个确定的初始值。但局部变量就不一样了，如果一个局部变量定义了但没有赋初始值是不能使用的。所以不要认为 Java 中任何情况下都存在诸如整型变量默认为 0、布尔型变量默认为 false 之类的默认值。正如下面的演示代码所示，这段代码其实并不能运行，所幸编译器能在编译期间检查到并提示这一点。即便编译器能通过手动生成字节码的方式制造出下面的代码效果，字节码检验的时候也会被虚拟机发现，从而导致类加载失败。

```
public  static void main(String[]  args)  {
int a;
System.out.println (a)j
)
```

12.2.3 操作数栈

操作数栈也被称为操作栈，是一个后入先出栈。同局部变量表一样，操作数栈的最大深度也在编译的时候被写入到 Code 属性的 max_stacks 数据项之中。操作数栈的每一个元素可以是任意的 Java 数据类型，包括 long 和 double。32 位数据类型所占的栈容量为 1，64 位数据类型所占的栈容量为 2。在方法执行的任何时候，操作数栈的深度都不会超过在 max-stacks 数据项中设定的最大值。

当一个方法刚刚开始执行的时候，这个方法的操作数栈是空的，在方法的执行过程中，会有各种字节码指令向操作数栈中写入和提取内容，也就是入栈出栈操作。例如在做算术运算的时候是通过操作数栈来进行的，又或者在调用其他方法的时候是通过操作数栈来进行参数传递的。

举个例子，整数加法的字节码指令 iadd 在运行的时候要求操作数栈中最接近栈顶的两个元素已经存入了两个 int 型的数值，当执行这个指令时，会将这两个 int 值出栈并相加，

然后将相加的结果入栈。

操作数栈中元素的数据类型必须与字节码指令的序列严格匹配，在编译程序代码的时候，编译器要严格保证这一点，在类校验阶段的数据流分析中还要再次验证这一点。再以上面的 iadd 指令为例，这个指令用于整型数加法，它在执行时，最接近栈顶的两个元素的数据类型必须为 int 型，不能出现一个 long 和一个 float 使用 iadd 命令相加的情况。

另外，在概念模型中，两个栈帧作为虚拟机栈的元素，相互之间是完全独立的。但是大多数虚拟机的实现里都会做一些优化处理，令两个栈帧出现一部分重叠。让下面栈帧的部分操作数栈与上面栈帧的部分局部变量表重叠在一起，这样在进行方法调用时就可以共用一部分数据，而无须进行额外的参数复制传递了。

Java 虚拟机的解释执行引擎称为“基于栈的执行引擎”，其中所指的“栈”就是操作数栈。

12.2.4　动态连接

每个栈帧都包含一个指向运行时常量池中该栈帧所属方法的引用，持有这个引用是为了支持方法调用过程中的动态连接。我们知道 Class 文件的常量池中存有大量的符号引用，字节码中的方法调用指令就以常量池中指向方法的符号引用为参数。这些符号引用一部分会在类加载阶段或第一次使用的时候转化为直接引用，这种转化称为静态解析。另外一部分将在每一次的运行期间转化为直接引用，这部分称为动态连接。

12.2.5　方法返回地址

当一个方法被执行后，有如下两种方式退出这个方法。

(1) 执行引擎遇到任意一个方法返回的字节码指令，这时候可能会有返回值传递给上层的方法调用者(调用当前方法的方法称为调用者)，是否有返回值和返回值的类型将根据遇到何种方法返回指令来决定，这种退出方法的方式称为正常完成出口(Normal Method Invocation Completion)。

(2) 在方法执行过程中遇到了异常，并且这个异常没有在方法体内得到处理，无论是 Java 虚拟机内部产生的异常，还是代码中使用 athrow 字节码指令产生的异常，只要在本方法的异常表中没有搜索到匹配的异常处理器，就会导致方法退出，这种退出方法的方式称为异常完成出口(Abrupt Method Invocation Completion)。

一个方法使用异常完成出口的方式退出，是不会给它的上层调用者产生任何返回值的。无论采用何种退出方式，在方法退出之后，都需要返回到方法被调用的位置，程序才能继续执行，方法返回时可能需要在栈帧中保存一些信息，用来帮助恢复它的上层方法的执行状态。一般来说，方法正常退出时，调用者的 PC 计数器的值就可以作为返回地址，栈帧中很可能会保存这个计数器值。而方法异常退出时，返回地址是要通过异常处理器表来确定的，栈帧中一般不会保存这部分信息。

方法退出的过程实际上等同于把当前栈帧出栈，因此退出时可能执行的操作有：恢复上层方法的局部变量表和操作数栈，把返回值(如果有的话)压入调用者栈帧的操作数栈中，调整 PC 计数器的值以指向方法调用指令后面的一条指令等。

12.2.6 附加信息

虚拟机规范允许具体的虚拟机实现增加一些规范里没有描述的信息到栈帧之中，例如与调试相关的信息，这部分信息完全取决于具体的虚拟机实现，这里不再详述。在实际开发中，一般会把动态链接、方法返回地址与其他附加信息全部归为一类，称为栈帧信息。

12.3 方法调用

方法调用并不等同于方法执行，方法调用阶段唯一的任务就是确定被调用方法的版本，即调用哪一个方法，暂时还不涉及方法内部的具体运行过程。在程序运行时，方法调用是最普遍、最频繁的操作。但是 Class 文件的编译过程中不包含传统编译中的连接步骤，一切方法调用在 Class 文件里面存储的都只是符号引用，而不是方法在实际运行时内存布局中的入口地址。这个特性给 Java 带来了更强大的动态扩展能力，但也使得 Java 方法的调用过程变得相对复杂起来，需要在类加载期间甚至到运行期间才能确定目标方法的直接引用。在本节的内容中，将详细讲解方法调用的基本知识。

12.3.1 方法调用的背景

程序在有限的资源下运行当然是越快越好，这就离不开优化。一般来说都是业务逻辑优化(这也是最有效的)，说到程序的运行的优化就不得不牵扯到 JVM 底层的字节码了。查看字节码的方法是 javap -c **.class，在此建议用保存成文本文件方便用工具查看。

```
javap -c **.class > **.txt
```

从 class 生成的字节码来看，Java 的方法调用分为 4 种： invokestatic、invokevirual、invokespecial、invokeinterface。

为了说明他们的区别，还得说下 JVM 中类的存储和方法的早/迟绑定。

1) Class 的类方法的存放地方和属性

JVM 有一个所有线程间共享的“方法区”，用来存储每个类结构的常数池、域、方法数据、方法和构造函数。包括类和实例初始化与接口类型初始化中用到的特殊方法。所以每当有线程调用某个类的方法时，都要从方法区调用。Java 类的方法有很多修饰，比如 static、final、private 等，这个决定了 jvm 在底层调用方法上的不同。

2) 方法的早/迟绑定

简单来说，要想分辨一个方法是早绑定还是迟绑定，可以通过是根据引用调用方法还是通过对象来调用方法来判断。当一个方法是 static 时，在任何地方都可以直接调用，比如 Math.abs(4)，这个时候就是早绑定，因为这里不需要用 new 新建，当然没对象了。早绑定不仅仅限于 static，当调用 private 方法时，也是早绑定，因为 private 只能被自身类方法调用。比如下面的 Java 代码：

```
class A{
  private void methodA(){}
```

```
}
class B extends A{
  private void methodB(){}
}
classA a = new classB();
```

此处无法用 a.methodA()和 a.methodB()，有了以上说明，就可以说清楚方法的具体区别了。

- invokestatic：用于 static 修饰的方法。任何时候调用只需要所属的 CLASS 名，无需 new，JVM 可以直接映射到方法区，是执行速度最快的。如果 static 方法有参数，则 invokestatic 指令前还会有个指令，作用是把参数从栈弹出给 invokestatic 指令。(在此没有详细讨论运行机制的问题，只是对 invokevirtual 等方法做了一个区分)
- invokevirtual：用于 public 和 protected 修饰的，且没有 static 修饰的方法，在 invokevirtual 之前，总可以见到 astore 和 aload 的指令，是因为在调用 invokevirtual 时，会从栈里弹出两个参数——objectref 和我们自定义的参数列表。objectref 就是 this，因为非 static 方法不是直接从方法区用的，所以得匹配所属类，是默认的隐式参数，无法从代码层指定 objectref。参数列表就是我们自定义的传入参数了。invokevirtual 是类的方法调用最慢的指令(因为迟绑定需要多重校验)，但是却是运用最多的，Java 面向对象的多态性离不开它。
- invokespecial：用于 3 种情况，前 2 种类默认的方法<init>()和 super 修饰的方法，它们可以为隐式，<init>()是默认的无参构造函数，super()是默认的调用父类构造函数的函数。当然我们也可以在代码层自定义些参数。invokespecial 可以默认从构造函数里递归调用 super()，而 invokevirtual 不行(动态绑定是只运行当前类中的方法)之前说过 invokespecial 是静态绑定的，如果换用动态绑定是会出错的(例如构造函数的例子)，第三种是非 static 修饰的 private 方法，原因之前也说了。invokespecial 的运行速度比较特殊，在 super 和 init()时，我们可以不用关心(关心也无用，改不了)，对于非 static 的 private 方法，速度是快于 invokevirtual 的。
- invokeinterface：用于接口调用的情况，速度是最慢的，因为接口不知道类的具体信息，所以每次运行前得遍历整个类(校验+匹配)，而 invokevirtual 是直接关联类的，方法偏移量是固定的。

首先，final 是用于不可修改，不可继承的用途，而不是改变使用或者说调用的方法。其次，对于代码优化规则来说，4 种运行速度为 invokestatic > invokespecial > invokevirual > invokeinterface，所以总结下常用方法是：

(1) 根据具体的业务要求，分离出常用的任务写成 static 方法，加快速度。有人也怀疑 static 方法会不会占用更多的内存，我认为不会，因为无论是什么样的方法都得占用方法区的空间，调用也是引用调用。再说对于现在上 G 的内存，我们写的几 K 的东西也算不上多大开销。

(2) 遵循高内聚、低耦合的模式，一个类只对外提供必要的 public 个 protected 方法，大部分的内部逻辑就用 private 修饰，一来速度快，二来也免得别人调用起来方法太多看得麻烦。

(3) 对于接口来说，不是用得越多越好，抽象出来的接口应该越精简越好，笔者的亲身体会是接口多了很麻烦，毕竟越灵活的东西越难理解。

12.3.2 解析

继续前面关于方法调用的话题，所有方法调用中的目标方法在 Class 文件里面都是一个常量池中的符号引用，在类加载的解析阶段，会将其中的一部分符号引用转化为直接引用，这种解析能成立的前提是：方法在程序真正运行之前就有一个可确定的调用版本，并且这个版本在运行期是不可改变的。换句话说，调用目标在程序代码写好、编译器进行编译时就必须确定下来。这类方法的调用称为解析(Resolution)。

在 Java 语言中，符合“编译期可知，运行期不可变”这个要求的方法主要有静态方法和私有方法两大类，前者与类型直接关联，后者在外部不可被访问，这两种方法都不可能通过继承或别的方式重写出其他版本，因此它们都适合在类加载阶段进行解析。

与之相对应，在 Java 虚拟机里面提供了 4 条方法调用字节码指令：

- invokestatic：调用静态方法。
- invokespecial：调用实例构造器<init>方法、私有方法和父类方法。
- invokevirtual：调用所有的虚方法。
- invokeinterface：调用接口方法，会在运行时再确定一个实现此接口的对象。

只要能被 invokestatic 和 invokespecial 指令调用的方法，都可以在解析阶段确定唯一的调用版本，符合这个条件的有静态方法、私有方法、实例构造器方法和父类方法四类，它们在类加载的时候就会把符号引用解析为该方法的直接引用。这些方法可以称为非虚方法，与之相反，其他方法就称为虚方法(除去 final 方法，后文会提到)。下面的代码演示了一个最常见的解析调用的例子，此样例中，静态方法 sayHello()只可能属于类型 StaticResolution，没有任何手段可以覆盖或隐藏这个方法。

```
public class StaticResolution {
    public static void sayHello() {
                          System. out .println ( " hello  world" )  ;
          }
     public static void main(String[] args) {
                     StaticResolution. sayHello ( ) ;
            }
      }
```

使用 javap 命令可以查看上述代码的字节码，查看后会发现是通过 invokestatic 命令来调用 sayHello()方法的。

```
D:\Develop\>javap -verbose StaticResolution
     public  static void main (java.lang.String [] ) ;
            Code :
   Stack=0, Locals=1, Args size-1
   0:    invokestatic    #31;  //Method sayHello: ()V
            3 :      return
```

```
LineNumberTable :
 line 15:0
 line  16 :   3
```

Java 中的非虚方法除了使用 invokestatic 和 invokespecial 调用的方法之外还有一种，就是被 final 修饰的方法。虽然 final 方法是使用 invokevirtual 指令来调用的，但是由于它无法被覆盖，没有其他版本，所以也无需对方法接收者进行多态选择，又或者说多态选择的结果肯定是唯一的。在 Java 语言规范中明确说明了 final 方法是一种非虚方法。

解析调用一定是个静态的过程，在编译期间就完全确定，在类装载的解析阶段就会把涉及的符号引用全部转变为可确定的直接引用，不会延迟到运行期再去完成。而分派(Dispatch)调用则可能是静态的也可能是动态的，根据分派依据的宗量数可分为单分派和多分派。这两类分派方式两两组合就构成了静态单分派、静态多分派、动态单分派、动态多分派四种分派情况，下面我们看看虚拟机中的方法分派是如何进行的。

12.3.3　分派

众所周知，Java 是一门面向对象的程序设计语言，因为 Java 具备面向对象的三个基本特征：继承、封装和多态。本节讲解的分派调用过程将会揭示多态性特征的一些最基本的体现(如“重载”和“重写”)，在 Java 中是如何实现的，这里的实现当然不是语法上该如何写，我们关心的依然是虚拟机如何确定正确的目标方法。

1. 静态分派

在开始讲解静态分派前，笔者准备了一段经常出现在面试题中的程序代码，读者不妨先看一遍，想一下程序的输出结果是什么。后面我们的话题将围绕这个类的方法来重载(Overload)代码，以分析虚拟机和编译器确定方法版本的过程。演示代码如下。

```
package  org.zzz.polymorphic;
public class StaticDispatch {
   static abstract class Human {
            }
   static class Man extends Human {
              }
   static class Women extends Human {
           }
   public void sayHello(Human guy) {
                 System.out.println ( " hello, guy ! " );
           }
   public void sayHello(Man guy) {
                 System.out.println ( " hello, gentlemanl ,, );
             }
  public void sayHello(Women guy) {
                 System.out.println ( "hello,lady ! " ) ;
          }
   public static void main(String[J args) {
         Human man = new Man();
                 Human women = new Women() ;
                 StaticDispatch sd = new StaticDispatch() ;
                 sr.sayHello (man) ;
```

```
                sr.sayHello (women) ;
            }
        }
```

运行结果：

```
hello, guy!
hello, guy!
```

上述演示代码实际上是在考察阅读者对重载的理解程度，相信对 Java 稍有经验的程序员看完程序后都能得出正确的运行结果，但为什么会选择执行参数类型为 Human 的重载呢？在解决这个问题之前，我们先使用如下代码定义两个重要的概念：

```
Human  man=new Man();
```

我们把上面代码中的“Human”称为变量的静态类型(Static Type)或者外观类型(Apparent Type)，后面的“Man”则称为变量的实际类型(Actual Type)，静态类型和实际类型在程序中都可以发生一些变化，区别是静态类型的变化仅仅在使用时发生，变量本身的静态类型不会被改变，并且最终的静态类型是在编译期可知的；而实际类型变化的结果在运行期才可确定，编译器在编译程序的时候并不知道一个对象的实际类型是什么。如下面的代码：

```
//实际类型变化
Human  man=new  Man();
man=new Women();
//静态类型变化
sr.sayHello( (Man)  man)
sr.sayHello( (Women)  man)
```

再次回到上述演示代码中。在方法 main()中进行了两次调用 sayHello()方法的过程，在方法接收者已经确定是对象“sr”的前提下，使用哪个重载版本，就完全取决于传入参数的数量和数据类型。代码中刻意地定义了两个静态类型相同、实际类型不同的变量，但虚拟机(准确地说是编译器)在重载时是通过参数的静态类型而不是实际类型作为判定依据的。并且静态类型是编译期可知的，所以在编译阶段，Javac 编译器就根据参数的静态类型决定使用哪个重载版本，所以选择了 sayHello(Human)作为调用目标，并把这个方法的符号引用写到 main()方法里的两条 invokevirtual 指令的参数中。

所有依赖静态类型来定位方法执行版本的分派动作，都称为静态分派。静态分派的最典型应用就是方法重载。静态分派发生在编译阶段，因此确定静态分派的动作实际上不是由虚拟机来执行的。另外，编译器虽然能确定出方法的重载版本，但在很多情况下这个重载版本并不是“唯一的”，往往只能确定一个“更加合适的”版本。这种模糊的结论在由 0 和 1 构成的计算机世界中算是个比较“稀罕”的事件，产生这种模糊结论的主要原因是字面量不需要定义，所以字面量没有显式的静态类型，它的静态类型只能通过语言上的规则去理解和推断。例如下面的代码演示了何为“更加合适的”版本。

```
package org.zzz.polymorphic;
public class Overload
  public static void sayHello(Object arg) {
             System.out .println ( "hello Object " ) ;
```

```
            }
    public static void sayHello(int arg) {
            System.out.println ( "hello  int " ) ;
        }
    public static void sayHello(long arg) {
            system. out .println ( " hello   long " )  ;
        }
    public static void sayHello(Character arg) {
            System.out .println ( "hello  Character" ) ;
        }
    public static void sayHello(char arg) {
            System. out .println ( " hello char" )  ;
        }
   public static void sayHello(char... arg) {
            System.out .println ( "hello  char …" ) ;
       }
    public static void sayHello(Serializable arg) {
            System. out .println ( " hello Serializable " )  ;
       }
    public static void main(String[] args) {
            sayHello ( 'a') ;
       }
}
```

运行后会输出：

```
hello char
```

由此可见，'a'是一个 char 类型的数据，自然会寻找参数类型为 char 的重载方法，如果注释掉 sayHello(char arg)方法，那么输出会变为：

```
hello int
```

这时发生了一次自动类型转换，'a'除了可以代表一个字符串外，还可以代表数字 65(字符'a'的 Unicode 数值为十进制数字 65)，因此参数类型为 int 的重载也是合适的。我们继续注释掉 sayHello(int arg)方法，那么输出会变为：

```
hello long
```

这时发生了两次自动类型转换，'a'转换为整数 65 之后，进一步转换为长整数 65L，匹配了参数类型为 long 的重载。笔者在代码中没有编写其他类型(如 float、double 等)的重载，不过实际上自动转型还能继续发生多次，按照 char—int—long—float—double 的顺序转型进行匹配。但不会匹配到 byte 和 short 类型的重载，因为 char 到 byte 或 short 的转型是不安全的。我们继续注释掉 sayHello(long arg)方法，那么输出会变为：

```
hello Character
```

这时发生了一次自动装箱，'a'被包装为它的封装类型 java.lang.Character，所以匹配到了参数类型为 Character 的重载，继续注释掉 sayHello(Character arg)方法，那么输出会变为：

```
hello Serializable
```

这个输出可能会让人摸不着头脑，一个字符或数字与序列化有什么关系？出现 hello Serializable，是因为 java.lang.Serializable 是 java.lang.Character 类实现的一个接口，自动装箱之后发现还是找不到装箱类，但是找到了装箱类实现了的接口类型，所以紧接着又发生一次自动转型。char 可以转型成 int，但是 Character 是绝对不会转型为 Integer 的，它只能安全地转型为它实现的接口或父类。Character 还实现了另外一个接口 java.lang.Comparable<Character>，如果同时出现两个参数分别为 Serializable 和 Comparable<Character>的重载方法，那么它们此时的优先级是一样的。编译器无法确定要自动转型为哪种类型，会提示类型模糊，拒绝编译。程序必须在调用时显式地指定字面量的静态类型，如 sayHello((Comparable<Character>)'a')，才能通过编译。下面继续注释掉 sayHello(Serializable arg)方法，输出会变为：

```
hello Object
```

这时是 char 装箱后转型为父类了，如果有多个父类，那么将在继承关系中从下往上开始搜索，越接近上层的优先级越低。即使方法调用传入的参数值为 null，这个规则仍然适用。我们把 sayHello(Object arg)也注释掉，输出将会变为：

```
hello char…
```

7 个重载方法已经被注释得只剩一个了，可见变长参数的重载优先级是最低的，这时候字符'a'被当作了一个数组元素。笔者使用的是 char 类型的变长参数，读者在验证时还可以选择 int 类型、Character 类型、Object 类型等的变长参数重载来把上面的过程重新演示一遍。但是要注意的是，有一些在单个参数中能成立的自动转型，如 char 转型为 int，在变长参数中是不成立的。

2. 动态分派

了解了静态分派，我们接下来看一下动态分派的过程，它和多态性的另外一个重要体现——重写(Override)有着很密切的关联。我们还是用前面的 Man 和 Women 一起，sayHello 的例子来讲解动态分派，请看下面的演示代码。

```
    public class DynamicDispatch {
    static abstract class Human {
                protected abstract void sayHello() ;
                      }
        static class Man extends Human {
@Override
            protected void sayHello() {
                System.out.println ( "man  say hello") ;
                      }
                   }
       static class Women extends Human {
@Override
            protected void sayHello() {
                System.out.println ( "women  say hello") ;
                      }
                   }
       public static void main(String[J args) {
            Human man - new Man();
```

```
                    Human women - new Women() ;
                      man. sayHello ( ) ;
                      women. sayHello ( )  ;
            . man = new Women();
                      man. sayHello ( ) ,
                    }
            }
```

运行结果：

```
man say hello
women say hello
women say hello
```

这个运行结果相信不会出乎任何人的意料，习惯了面向对象思维的 Java 程序员会觉得这是完全理所当然。我们现在的问题还是和前面的一样，虚拟机是如何知道要调用哪个方法的？

显然这里是不可能根据静态类型来决定的，因为静态类型都是 Human 的两个变量 man 和 women 在调用 sayHello()方法时执行了不同的行为，并且变量 man 在两次调用中执行了不同的方法。导致这个现象的原因很明显，是这两个变量的实际类型不同，Java 虚拟机是如何根据实际类型来分派方法执行版本的呢？我们使用 javap 命令输出这段代码的字节码，结果如下面的演示代码：

```
    public  static  void main (java.lang.String [] ) ;
            Code :
   Stack=2, Locals=3, Args size=l
     0:   new    #16; //class
  org/zzz/p olymorphic/DynamicDispatch$Man
     3 :     dup
   4:  invokespecial  #18; //Method
 org/zzz/polymorphic/DynamicDispatch$Man. " <init > " :  ( ) V
            7 :      astore l
    8:   new   #19; //class
org/zzz/polymorphic/DynamicDispatch$Women
                     11 :       dup
   12:  invokespecial  #21; //Method
 org/zzz/polymorphic/DynamicDispatch$Women. " <init > " :  ( ) V
   15 :    astore 2
   16:  aload l
   17:  invokevirtual  #22; //Method
 org/f enixsoft/polymorphic/DynamicDispatch$Human . sayHello :  ( ) v
   20:  aload 2
   21:  invokevirtual  #22; //Method
org/zzz/polymorphic/DynamicDispatch$Human . sayHello :  ( ) v
    24:  new   #19; //class
 org/f enixs o f t /polymorphic/DynamicDispat ch$Women
               2 7 :     dup
   28:  invokespecial  #21; //Method
org/zzz/polymorphic/DynamicDispatch$Women. ri<init > " :  ( ) v
   31:  astore l
   32:  aload 1
   33:  invokevirtual   #22. //Method
```

```
org/zzz/polymorphic/DynamicDispat ch$Human. sayHello :  ( ) V
    3 6 :     return
```

在上述代码中，0～15 行的字节码是准备动作，作用是建立 man 和 women 的内存空间、调用 Man 和 Women 类型的实例构造器，将这两个实例的引用存放在第 1 和第 2 个局部变量表 Slot 之中，这个动作对应了代码中的这两句：

```
Human man=new Man();
Human women=new Women();
```

接下来的第 16～21 行是关键部分，第 16 和第 20 两行分别把刚刚创建的两个对象的引用压到栈顶，这两个对象是将要执行的 sayHello()方法的所有者，称为接收者(Receiver)；第 17 和第 21 两行是方法调用指令，单从字节码的角度来看，这两条调用指令无论是指令(都是 invokevirtual)还是参数(都是常量池中第 22 项的常量，注释显示了这个常量是 Human.sayHello()的符号引用)都完全一样，但是这两条指令最终执行的目标方法并不相同，其原因需要从 invokevirtual 指令的多态查找过程开始说起，invokevirtual 指令的运行时解析过程大致分为以下几步：

(1) 找到操作数栈顶的第一个元素所指向的对象的实际类型，记作 C。

(2) 如果在类型 C 中找到与常量中的描述符和简单名称都相符的方法，则进行访问权限校验，如果通过则返回这个方法的直接引用，查找过程结束；不通过则返回 java.lang.IllegalAccessError 异常。

(3) 否则，按照继承关系从下往上依次对 C 的各个父类进行第 2 步的搜索和验证过程。

(4) 如果始终没有找到合适的方法，则抛出 java.lang.AbstractMethodError 异常。

由于 invokevirtual 指令执行的第一步就是在运行期确定接收者的实际类型，所以两次调用中的 invokevirtual 指令把常量池中的类方法符号引用解析到了不同的直接引用上，这个过程就是 Java 语言中方法重写的本质。我们把这种在运行期根据实际类型确定方法执行版本的分派过程称为动态分派。

3. 单分派与多分派

方法的接收者与方法的参数统称为方法的宗量，这个定义最早应该来源于《Java 与模式》一书的译文。根据分派基于多少种宗量，可以将分派划分为单分派和多分派两种。单分派是根据一个宗量对目标方法进行选择，多分派则是根据多于一个的宗量对目标方法进行选择。

单分派和多分派的定义相当拗口，不过对照着实例看就不难理解了，在下面的演示代码中列举了一个 Baba 和 Erzi 一起来做出决定的例子。

```
public class Dispatch {
static class QQ {}
   static class 360 {}
   public static class Baba {
      public void hardChoice(QQ arg) {
                        System.out .println ( " baba  choose  qq") ;
                  }
     public void hardChoice( 360 arg) {
```

```
                    System.out.println ( " baba  choose  360 ") ;
               }
         }
     public static class Erzi ex.tends Baba {
          public void hardChoice(QQ arg) {
                    System.out.println ( "erzi  choose  qq") ;
                    }
          public void hardChoice( 360 arg) {
                    System.out.println ( "erzi  choose  360 ") ;
                    }
          }
     public static void main(String[l args) {
               Baba baba = new Baba() ;
          Baba sori = new Erzi();
                    baba.hardChoice (new   360 ( ) ) ;
                    erzi.hardChoice (new  QQ () ) ;
          }
     }
```

运行结果：

```
baba choose 360
erzi choose qq
```

在 mam 函数中调用了两次 hardChoice()方法，这两次 hardChoice()方法的选择结果在程序输出中已经显示得很清楚了。

再来看看编译阶段编译器的选择过程，即静态分派的过程。这时候选择目标方法的依据有两点：一是静态类型是 Baba 还是 Erzi，二是方法参数是 QQ 还是 360。这次选择结果的最终产物是产生了两条 invokevirtual 指令，两条指令的参数分别为常量池中指向 Baba.hardChoice(360)及 Baba.hardChoice(QQ)方法的符号引用。因为是根据两个宗量进行选择，所以 Java 语言的静态分派属于多分派类型。

再看看运行阶段虚拟机的选择，即动态分派的过程。在执行“erzi.hardChoice(new QQ())”这句代码时，更准确地说，在执行这句代码所对应的 invokevirtual 指令时，由于编译期已经决定目标方法的签名必须为 hardChoice(QQ)，虚拟机此时不会关心传递过来的参数“QQ”到底是“腾讯 QQ”还是“奇瑞 QQ”，因为这时候参数的静态类型、实际类型都不会对方法的选择构成任何影响，唯一可以影响虚拟机选择的因素只有此方法的接收者的实际类型是 Baba 还是 Erzi。因为只有一个宗量作为选择依据，所以 Java 语言的动态分派属于单分派类型。

根据上述论证的结果可以看出，当前的 Java 语言是一门静态多分派、动态单分派的语言。强调“当前的 Java 语言”是因为这个结论未必会恒久不变，C#在 3.0 及之前的版本与 Java 一样是动态单分派语言。但是在 C# 4.0 中引入了 dynamic 类型后，就可以很方便地实现动态多分派。Java 也已经在 JSR-292 中开始规划对动态语言的支持了，日后将有可能提供类似的动态类型功能。

4．虚拟机动态分派的实现

前面介绍的分派过程，作为对虚拟机概念模型的解析基本上已经足够了，它已经解决

了虚拟机在分派中“会做什么”这个问题。但是虚拟机“具体是如何做到的”，可能各种虚拟机的实现都会有所差别。

由于动态分派是非常频繁的动作，而且动态分派的方法版本选择过程需要运行时在类的方法元数据中搜索合适的目标方法，因此在虚拟机的实际实现中基于性能的考虑，大部分实现都不会真的进行如此频繁的搜索。面对这种情况，最常用的“稳定优化”手段就是为类在方法区中建立一个虚方法表(Virtual Method Table，也称为 vtable，与此对应，在 invokeinterface 执行时也会用到接口方法表-Interface Method Table，简称 itable)，使用虚方法表索引来代替元数据查找以提高性能。我们先看看代码清单 8-10 所示。

虚方法表中存放着各个方法的实际入口地址。如果某个方法在子类中没有被重写，那么子类的虚方法表里面的地址入口和父类相同方法的地址入口是一致的，都指向父类的实现入口。如果子类中重写了这个方法，子类方法表中的地址将会被替换为指向子类实现版本的入口地址。

为了程序实现上的方便，具有相同签名的方法，在父类、子类的虚方法表中都应当具有一样的索引序号，这样当类型变换时，仅需要变更查找的方法表，就可以从不同的虚方法表中按索引转换出所需的入口地址。

方法表一般在类加载的连接阶段进行初始化，准备了类的变量初始值后，虚拟机会把该类的方法表也初始化完毕。

上文中笔者说方法表是分派调用的“稳定优化”手段，虚拟机除了使用方法表之外，在条件允许的情况下，还会使用内联缓存(Inline Cache)和基于“类型继承关系分析”(Class Hierarchy Analysis，CHA)技术的守护内联(Guarded Inlining)两种非稳定的“激进优化”手段来获得更高的性能，关于这两种优化技术的原理和运作过程，读者可以参考本书第 11 章中的相关内容。

12.4 基于栈的字节码解释执行引擎

关于虚拟机是如何调用方法已经讲解完毕，从本节开始，我们来探讨虚拟机是如何执行方法里面的字节码指令的。概述中提到过，许多 Java 虚拟机的执行引擎在执行 Java 代码的时候都有解释执行(通过解释器执行)和编译执行(通过即时编译器产生本地代码执行)两种选择，下面我们先来探讨一下在解释执行时虚拟机执行引擎是如何工作的。

12.4.1 解释执行

Java 语言经常被人们定位为“解释执行”的语言，在 Java 初生的 JDK 1.0 时代，这种定义还算是比较准确的，但当主流的虚拟机中都包含了即时编译器后，Class 文件中的代码到底会被解释执行还是编译执行，就成了只有虚拟机自己才能准确判断的事。再后来，Java 发展出了可以直接生成本地代码的编译器(如 GNU Compiler for the Java，GCJ)，而 C/C++语言也出现了通过解释器执行的版本(如 CINT)，这时候再笼统地说“解释执行”对于整个 Java 语言来说几乎就是没有意义的概念了，只有确定了谈论对象是某种具体的 Java

实现版本和执行引擎运行模式时，谈解释执行还是编译执行才会比较确切。

不论是解释还是编译，也不论是物理机还是虚拟机，对于应用程序，机器都不可能如人那样阅读和理解，然后就获得了执行能力。大部分的程序代码到物理机的目标代码或虚拟机能执行的指令集之前，都需要经过下面的各个步骤。

(1) 程序源码。

(2) 分析词法。

(3) 单词流。

(4) 分析语法。

(5) 抽象语法树：此过程有两个分支。

第一个分支如下：

- 指令流(可选的)。
- 解释器。
- 解释执行。

第二个分支如下：

- 中间代码(可选的)。
- 生成器。
- 目标代码。

如今，基于物理机、Java 虚拟机或者是非 Java 的其他高级语言虚拟机(HLLVM)的语言，大多都遵循这种基于现代经典编译原理的思路，在执行前先对程序源码进行词法分析和语法分析处理，把源码转化为抽象语法树(Abstract Syntax Tree，AST)。对于一门具体语言的实现来说，词法和语法分析乃至后面的优化器和目标代码生成器都可以选择独立于执行引擎，形成一个完整意义的编译器去实现，这类代表是 C/C++语言。也可以选择把其中一部分步骤(如生成抽象语法树之前的步骤)实现为一个半独立的编译器，这类代表是 Java 语言。又或者把这些步骤和执行引擎全部集中封装在一个封闭的黑匣子之中，如大多数的 JavaScript 执行器。

Java 语言中，Javac 编译器完成了程序代码经过词法分析、语法分析到抽象语法树，再遍历语法树生成线性的字节码指令流的过程。因为这一部分动作是在 Java 虚拟机之外进行的，而解释器在虚拟机的内部，所以 Java 程序的编译就是半独立的实现。

12.4.2 基于栈的指令集与基于寄存器的指令集

Java 编译器输出的指令流，基本上是一种基于栈的指令集架构(Instruction Set Architecture，ISA)，指令流里面的指令大部分都是零地址指令，它们依赖操作数栈进行工作。与之相对的另外一套常用的指令集架构是基于寄存器的指令集，最典型的就是 x86 的二地址指令集，更通俗一些，就是现在我们主流 PC 中直接支持的指令集架构，这些指令依赖寄存器进行工作。那么，基于栈的指令集与基于寄存器的指令集这两者之间有什么不同呢？

举个最简单的例子，分别使用这两种指令集去计算“1+1”的结果，基于栈的指令集会是这样子的：

```
Iconst 1
iconst 1
iadd
istore_0
```

两条 iconst 1 指令连续地把两个常量 1 压入栈后，iadd 指令把栈顶的两个值出栈并相加，然后把结果放回栈顶，最后 istore_0 把栈顶的值放到局部变量表的第 0 个 Slot 中。

如果是基于寄存器的指令集，那么程序可能会是这个样子的：

```
mov  eax,1
+add  eax,  1
```

mov 指令把 EAX 寄存器的值设为 1，然后 add 指令再把这个值加 1，结果就保存在 EAX 寄存器里面。

了解了基于栈的指令集与基于寄存器的指令集的区别后，读者可能会有进一步的疑问，这两套指令集谁更好一些呢？应该说，既然两套指令集同时并存和发展，那么肯定是各有优势的，如果有一套指令集全面优于另外一套的话，就不存在选择的问题。

基于栈的指令集最主要的优点就是可移植性，寄存器由硬件直接提供，程序直接依赖这些硬件寄存器则不可避免地要受到硬件的约束。例如，现在 32 位 x86 体系的处理器中提供了 8 个 32 位的寄存器，而 ARM 体系的 CPU(在当前的手机、PDA 中相当流行的一种处理器)则提供了 16 个 32 位的通用寄存器。如果使用栈架构的指令集，用户程序不会直接用到这些寄存器，那就可以由虚拟机实现来自行决定把一些访问最频繁的数据(程序计数器、栈顶缓存等)放到寄存器中以获取尽量好的性能，这样实现起来也更加简单。栈架构的指令集还有一些其他优点，如代码相对更紧凑(字节码中每个字节就对应一条指令，而多地址指令集中还需要存放参数)、编译器实现更加简单(不需要考虑空间分配的问题，所需空间都在栈上操作)等。

栈架构指令集的主要缺点是执行速度相对来说稍慢一些。栈架构指令集的代码虽然紧凑，但是完成相同功能所需的指令数量一般会比存器架构多，因为出栈、入栈操作本身就产生了相当多的指令。更重要的是栈实在内存之中，频繁的栈访问也就意味着频繁的内存访问，相对于处理器来说，内始终是执行速度的瓶颈。尽管虚拟机可以采取栈顶缓存的手段，把最常用的操作射到寄存器中以避免直接内存访问，但这也只能是优化措施而不是解决本质问题方法。因此，由于指令数量和内存访问的原因，导致了栈架构指令集的执行速度相对较慢。

12.4.3　基于栈的解释器执行过程

初步的理论已经讲解完了，本小节准备了一段 Java 代码，看看在虚拟机中实际上是如何执行的。例如在下面的演示代码中，展示了四则运算的过程。

```
public class Demo {
   public static void foo() {
      int a = 1;
      int b = 2;
      int c = (a + b) * 5;
   }
```

```
}
```

这段代码从 Java 语言的角度没有任何评论的必要，直接使用 javap 命令看看它的字节码指令。通过 javac 编译，可以得到文件 Demo.class。通过 javap 可以看到 foo()方法的字节码如下：

```
1.0:  iconst 1
2.1:  istore 0
3.2:  iconst 2
4.3:  istore 1
5.4:  iload 0
6.5:  iload 1
7.6:  iadd
8.7:  iconst 5
9.8:  imul
10.9:  istore 2
11.10: return
0:  iconst 1
1:  istore 0
2:  iconst 2
3:  istore 1
4:  iload 0
5:  iload 1
6:  iadd
7:  iconst 5
8:  imul
9:  istore 2
10: return
```

这说明 Java 字节码以 1 字节为单元。上面代码中有 11 条指令，每条都只占 1 单元，共 11 单元==11 字节。程序计数器是用于记录程序当前执行的位置用的。对 Java 程序来说，每个线程都有自己的 PC。PC 以字节为单位记录当前运行位置里方法开头的偏移量。

每个线程都有一个 Java 栈，用于记录 Java 方法调用的“活动记录”(Activation Record)。Java 栈以帧(Frame)为单位线程的运行状态，每调用一个方法就会分配一个新的栈帧压入 Java 栈上，每从一个方法返回则弹出并撤销相应的栈帧。

每个栈帧包括局部变量区、求值栈(JVM 规范中将其称为“操作数栈”)和其他一些信息。局部变量区用于存储方法的参数与局部变量，其中参数按源码中从左到右顺序保存在局部变量区开头的几个 slot。求值栈用于保存求值的中间结果和调用别的方法的参数等。两者都以字长(32 位的字)为单位，每个 slot 可以保存 byte、short、char、int、float、reference 和 returnAddress 等长度小于或等于 32 位类型的数据；相邻两项可用于保存 long 和 double 类型的数据。每个方法所需要的局部变量区与求值栈大小都能够在编译时确定，并且记录在.class 文件里。

在上面的例子中，Demo.foo()方法所需要的局部变量区大小为 3 个 slot，需要的求值栈大小为 2 个 slot。Java 源码的 a、b、c 分别被分配到局部变量区的 slot 0、slot 1 和 slot 2。可以观察到 Java 字节码是如何指示 JVM 将数据压入或弹出栈，以及数据是如何在栈与局部变量区之前流动的；可以看到数据移动的次数特别多。动画里可能不太明显，iadd 和 imul 指令都是要从求值栈弹出两个值运算，再把结果压回到栈上的；光这样一条指令就有

3 次概念上的数据移动了。

Java 的局部变量区并不需要把某个局部变量固定分配在某个 slot 里。不仅如此，在一个方法内某个 slot 甚至可能保存不同类型的数据。如何分配 slot 是编译器的自由。从类型安全的角度看，只要对某个 slot 的一次 load 的类型与最近一次对它的 store 的类型匹配，JVM 的字节码校验器就不会抱怨。因为这方面的内容比较高深，读者读懂需要很广的知识面，所以在本书中不再详细讲解。



第 13 章 类加载器和执行子系统

在 Class 文件格式与执行引擎这部分里，用户的程序能直接影响的内容并不太多，Class 文件以何种格式存储，类型何时加载、如何连接，以及虚拟机如何执行字节码指令等都是由虚拟机直接控制的行为，用户程序无法对其进行改变。能通过程序进行操作的，主要是字节码生成与类加载器这两部分的功能，但仅仅在如何处理这两点上，就已经出现了许多值得欣赏和借鉴的思路，这些思路后来成为了许多常用功能和程序实现的基础。本章将详细讲解各种类加载器和执行子系统的基本知识，为读者学习后面的知识打下基础。

13.1 分析 Tomcat 类加载器的架构

主流的 Java Web 服务器，例如 Tomcat、Jetty、WebLogic、WebSphere 都实现了自己定义的类加载器。因为一个功能健全的 Web 服务器，都要解决如下几个问题。

(1) 部署在同一个服务器上的两个 Web 应用程序所使用的 Java 类库可以实现相互隔离。这是最基本的需求，两个不同的应用程序可能会依赖同一个第三方类库的不同版本，不能要求一个类库在一个服务器中只有一份，服务器应当可以保证两个应用程序的类库可以互相独立使用。

(2) 部署在同一个服务器上的两个 Web 应用程序所使用的 Java 类库可以互相共享。这个需求也很常见，例如用户可能有 10 个使用 Spring 组织的应用程序部署在同一台服务器上，如果把 10 份 Spring 分别存放在各个应用程序的隔离目录中，将会是很大的资源浪费——这主要倒不是浪费磁盘空间的问题，而是指类库在使用时都要被加载到服务器内存，如果类库不能共享，虚拟机的方法区很容易就会出现过度膨胀的风险。

(3) 服务器需要尽可能地保证自身的安全不受部署的 Web 应用程序影响。目前，有许多主流的 Java Web 服务器自身也是使用 Java 语言来实现的。因此服务器本身也有类库依赖的问题，一般来说，基于安全考虑，服务器所使用的类库应该与应用程序的类库互相独立。

(4) 支持 JSP 应用的 Web 服务器，十有八九都需要支持 HotSwap 功能。我们知道 JSP 文件最终要被编译成 Java Class 才能被虚拟机执行，但 JSP 文件由于其纯文本存储的特性，被运行时修改的概率远远大于第三方类库或程序自己的 Class 文件。而且 ASP、PHP 和 JSP 这些网页应用也把修改后无须重启作为一个很大的“优势”来看待，因此“主流”的 Web 服务器都会支持 JSP 生成类的热替换，当然也有“非主流”的，如运行在生产模式(Production Mode)下的 WebLogic 服务器默认就不会处理 JSP 文件的变化。

由于存在上述问题，在部署 Web 应用时，单独的一个 ClassPath 就无法满足需求了，所以各种 Web 服务器都提供了好几个 ClassPath 路径供用户存放第三方类库，这些路径一般都以“lib”或“classes”命名。被放置到不同路径中的类库，具备不同的访问范围和服务对象，通常，每一个目录都会有一个相应的自定义类加载器去加载放置在里面的 Java 类库。现在，笔者就以 Tomcat 服务器为例，看一看 Tomcat 具体是如何规划用户的类库结构和类加载器的。

13.1.1 Tomcat 目录结构

在 Tomcat 目录结构中，有三组目录(“/common/*”、“/server/*”和“/shared/*”)可以存放 Java 类库，另外还可以加上 Web 应用程序自身的目录“/WEB-INF/*”，一共 4 组，把 Java 类库放置在这些目录中的含义分别是：

- 放置在/common 目录中：类库可被 Tomcat 和所有的 Web 应用程序共同使用。
- 放置在/server 目录中：类库可被 Tomcat 使用，对所有的 Web 应用程序都不可见。

- 放置在/shared 目录中：类库可被所有的 Web 应用程序共同使用，但对 Tomcat 自己不可见。
- 放置在/WebApp/WEB-INF 目录中：类库仅仅可以被此 Web 应用程序使用，对 Tomcat 和其他 Web 应用程序都不可见。

为了支持这套目录结构，并对目录里面的类库进行加载和隔离，Tomcat 自定义了多个类加载器，这些类加载器按照经典的双亲委派模型来实现，其关系分别如图 13-1 和图 13-2 所示。

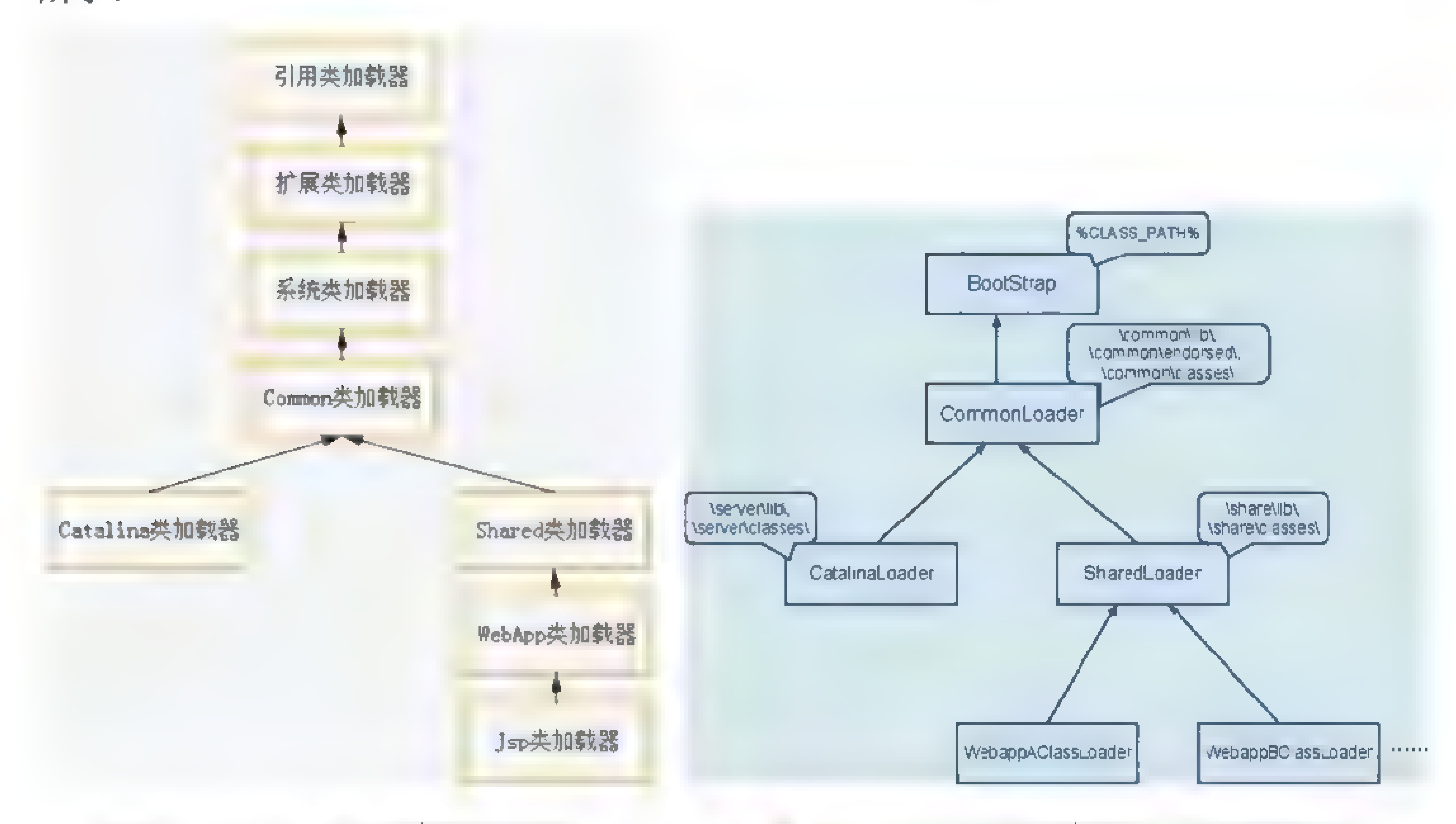


图 13-1　Tomcat 类加载器的架构　　　　图 13-2　Tomcat 类加载器的文件架构结构

图 13-2 展示了各层类加载器以及类文件搜索路径，Tomcat 为每个部署到其中的 Web 项目定义一个类加载器(如上图中 WebappAClassLoader、WebappBClassLoader)，其类文件搜索路径即为%CATALINA_HOME%\webapps\项目名称\WEB-INF\lib\;%CATALINA_HOME%\webapps\项目名称\WEB-INF\classes\。Tomcat 自己定义了一个 BootStrap 类，在 org.apache.catalina.startup.BootStrap 定义。

CommonClassLoader、CatalinaClassLoader、ShareClassLoader、和 WebappClassLoader 是 Tomcat 自定义加载器，分别对应加载/common/*、/server/*、/shared/*和 /WEB-INF/*类库，其中 Webapp 类加载器和 Jsp 类加载器会存在多个，每个 Web 应用对应一个 Webapp 类加载器，每个 JSP 文件对应 Jsp 类加载器。

CommonClassLoader 加载的类可以被 CatalinaClassLoader 和 ShareClassLoader 使用；CatalinaClassLoader 加载的类和 ShareClassLoader 加载的类相互隔离；WebappClassLoader 可以使用 ShareClassLoader 加载的类，但各个 WebappClassLoader 间相互隔离；JspClassLoader 仅能用 JSP 文件编译的 Class 文件。

13.1.2 定义公共类加载器

在 Tomcat 中，可以通过如下代码定义类加载器。

```
private Object catalinaDaemon = null;//定义 catalina 服务器守护程序实例
protected ClassLoader commonLoader = null;
protected ClassLoader catalinaLoader = null;
protected ClassLoader sharedLoader = null;

private void initClassLoaders() {
    try {
        ClassLoaderFactory.setDebug(debug);
        commonLoader = createClassLoader("common", null);
        catalinaLoader = createClassLoader("server", commonLoader);
        sharedLoader = createClassLoader("shared", commonLoader);
    } catch (Throwable t) {
        log("Class loader creation threw exception", t);
        System.exit(1);
    }
}
```

我们看到 commonLoader 的父类加载器为 null，即在委派机制下它将把类加载任务直接委派给 JVM 所使用的 BootStrap Loader，但为什么是 null 呢？因为 JVM 所使用的 BootStrap Loader 是用 C++编写的。

catalinaDaemon 为服务器从启动至停止都存在的守护线程，类似于桌面程序中的 Main 守护线程。createClassLoader 函数利用 ClassLoaderFactory 类在工厂模式下创建，创建代码如下：

```
public static ClassLoader createClassLoader(File unpacked[],File
packed[], URL urls[], ClassLoader parent)throws Exception {
..
      //获得将要创建的类加载器的类文件搜索路径
      String array[] = (String[]) list.toArray(new String[list.size()]);
      StandardClassLoader classLoader = null;
      if (parent == null)//父加载器为 JVM 使用的 BootStrap Loader
         classLoader = new StandardClassLoader(array);
      else
         classLoader = new StandardClassLoader(array, parent);
      classLoader.setDelegate(true);//设置该类加载器遵循委派模式
      return (classLoader);
```

Tomcat 提供两种类加载器供使用：一种是如上代码中所述的标准类加载器 StandardClassLoader，用以实例化为 commonLoader、catalinaLoader 和 sharedLoader 在 org.apache.catalina.loader.StandardClassLoader 中定义，它不提供热部署功能；另外一种是专为 Web 程序所提供的 WebClassLoader，它用以实例化为各部署项目的类加载器，在 org.apache.catalina.loader.WebappClassLoader 中定义，提供热部署功能，也就是在发生 ClassLoader 搜索路径下的资源改变的动作之后，服务器自动重新加载之。

13.1.3　初始化 catalina 守护程序

下面是 Tomcat 初始化 catalina 守护程序的具体代码。

```
public void init()
    throws Exception
{
    // Set Catalina path
    setCatalinaHome();
    setCatalinaBase();

    initClassLoaders();
    Thread.currentThread().setContextClassLoader(catalinaLoader);
    SecurityClassLoad.securityClassLoad(catalinaLoader);
    /*利用类加载器 catalinaClassLoader 加载 Catalina，并调用后者的 process 方法，
该方法设置%CATALINA HOME%,%CATALINA BASE%，并根据参数配置启动 catalina*/
    Class startupClass =
        catalinaLoader.loadClass
        ("org.apache.catalina.startup.Catalina");
    Object startupInstance = startupClass.newInstance();
    /*将 SharedClassLoader 设为 Catalina 类的 ClassLoader*/
    String methodName = "setParentClassLoader";
    Class paramTypes[] = new Class[1];
    paramTypes[0] = Class.forName("java.lang.ClassLoader");
    Object paramValues[] = new Object[1];
    paramValues[0] = sharedLoader;
    Method method =
        startupInstance.getClass().getMethod(methodName, paramTypes);
    method.invoke(startupInstance, paramValues);
    catalinaDaemon = startupInstance;
}
```

本段代码实现了利用 CatalinaClassLoader 加载 Catalina 类，并创建其实例。有意思的是，创建实例之后 Catalina 的类加载器却被设置为 SharedClassLoader。

我们知道 Catalina 是 Tomcat 容器的代言人，也就是一个在容器生命周期内都存在的类，我们所设计的 Servlet 是被放置在这个容器里面供调用的，从代码层来讲也就是被实例化，然后引用。同时，在 Java 类加载器体系结构中定义到：被引用类默认由依赖类的 ClassLoader 加载，而这样设计的原因是，运行时相同层次的 ClassLoader 所加载的类无法看到其他 ClassLoader 所加载的类，可这又是为什么呢？这是 Java 语言的安全特性所要求的。由上所述，可以知道如果要引用 Servlet 的话，得由 Catalina 的 ClassLoader 去加载，但是我们之前已经看到了，每一个 Web 项目都有一个特定的 WebappClassLoder 加载，并且 Catalina 需要引用的可是同时部署到其中的许多个 Web 项目的 Servlet，这就出现了矛盾。但是，我们仔细看看 WebappClassLoader 的设计，它的父加载器是 SharedClassLoader，SharedClassLoader 加载的是部署到容器中的多个 Web 项目共用的资源，所以将 Catalina 的类加载器设置为 SharedClassLoader，这样利用父加载器加载 Catalina，而用子加载器来加载 Served。

13.1.4　Tomcat 内部初始化类加载器

在初步了解了 Tomcat 关于类加载器的一些知识后，接下来详细看看 Tomcat 内部是怎么来初始化这些类加载器的。首先，Java 程序都需要一个入口(main 方法)，而在 tomcat 中，这个入口在 org.apache.catalina.startup.Bootstrap 这个类中，其结构如图 13-3 所示。

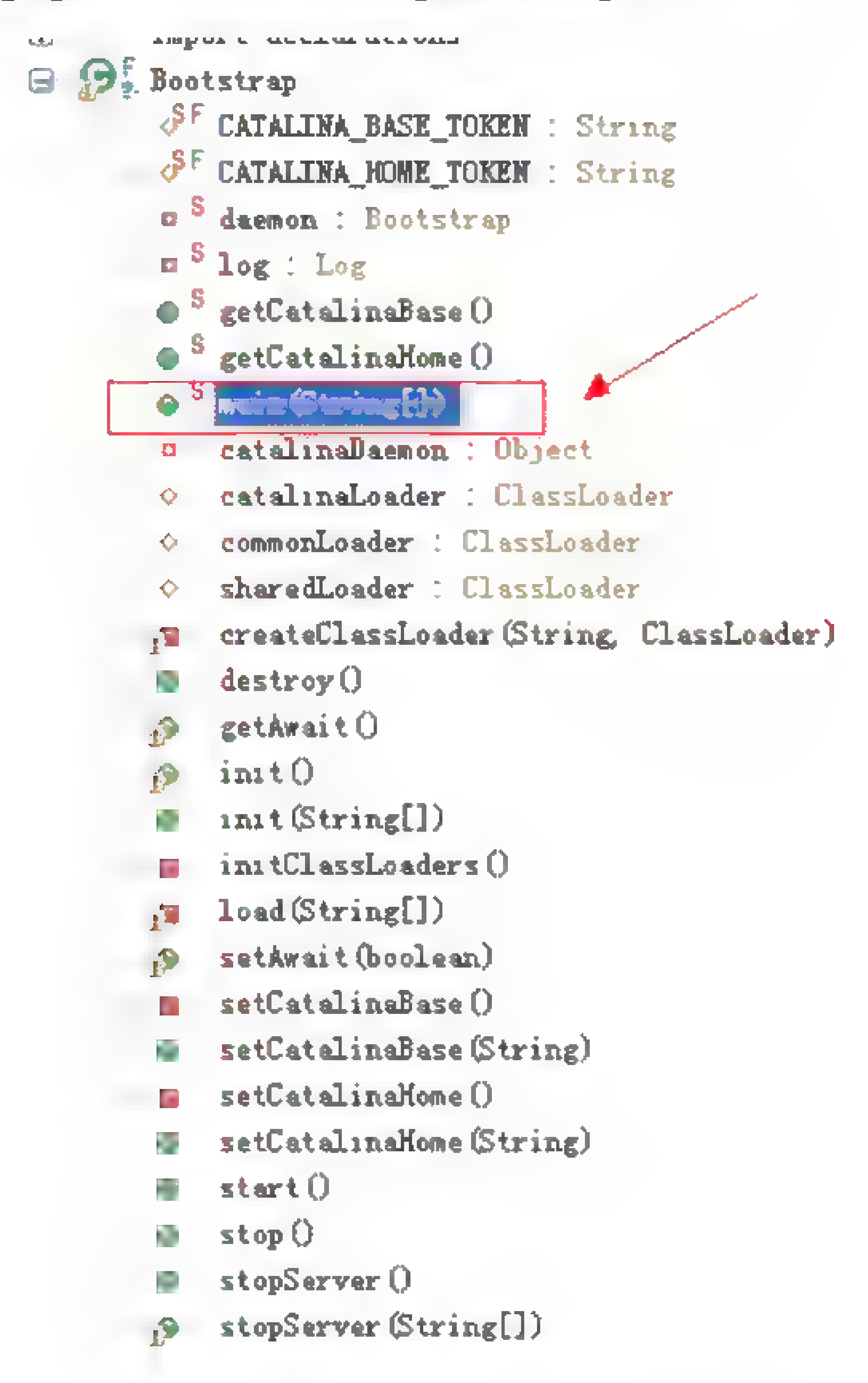


图 13-3　类 Bootstrap 的结构

如果定位到方法内部，其代码如下。

```
public static void main(String args[]) {
      if (daemon == null) {
         daemon = new Bootstrap();
         try {
             //初始化资源  (今天来了解的.)
            daemon.init();
         } catch (Throwable t) {
            t.printStackTrace();
            return;
         }
      }

      try {
```

```
        //默认为启动
        String command = "start";
        if (args.length > 0) {
           command = args[args.length - 1];
        }

        if (command.equals("startd")) {
           args[args.length - 1] = "start";
           daemon.load(args);
           daemon.start();
        } else if (command.equals("stopd")) {
           args[args.length - 1] = "stop";
           daemon.stop();
        } else if (command.equals("start")) {
            //设置标识
           daemon.setAwait(true);
           daemon.load(args);
           //开启
           daemon.start();
        } else if (command.equals("stop")) {
           daemon.stopServer(args);
        } else {
           log.warn("Bootstrap: command \"" + command + "\" does not exist.");
        }
     } catch (Throwable t) {
        t.printStackTrace();
     }
  }
```

在 Tomcat 启动之前，需要初始化一些系统资源，初始化的详细工作都定义在 init()方法内部了。

接下来继续追踪，定位到 init()方法中，代码如下。

```
  public void init()
     throws Exception
  {
     // Set Catalina path 设置 catalina 基本路径
     setCatalinaHome();
     setCatalinaBase();

     //初始化类加载器
     initClassLoaders();

     Thread.currentThread().setContextClassLoader(catalinaLoader);
     SecurityClassLoad.securityClassLoad(catalinaLoader);
     // Load our startup class and call its process() method
     if (log.isDebugEnabled())
        log.debug("Loading startup class");
     Class startupClass =
catalinaLoader.loadClass("org.apache.catalina.startup.Catalina");
     Object startupInstance = startupClass.newInstance();
     // Set the shared extensions class loader
     if (log.isDebugEnabled())
        log.debug("Setting startup class properties");
```

```
        String methodName = "setParentClassLoader";
        Class paramTypes[] = new Class[1];
        paramTypes[0] = Class.forName("java.lang.ClassLoader");
        Object paramValues[] = new Object[1];
        paramValues[0] = sharedLoader;
        Method method =startupInstance.getClass().getMethod(methodName,
paramTypes);
        method.invoke(startupInstance, paramValues);
        catalinaDaemon = startupInstance;
    }
```

由此可以看到，上面的代码中，用来初始化类加载器、验证类加载器，以及使用类加载器来加载类 org.apache.catalina.startup.Catalina 等操作。Tomcat 会调用 initClassLoaders()方法，用来初始化 common、catalina 和 shared 三种类加载器，而这个操作是通过方法 createClassLoader(String name, ClassLoader parent)来完成的，而后两个都属于 common 的子级，所以下面给出两个方法的源代码。

```
/**
     * 初始化类加载器：
     * 加载三种：
     *        common.
     *        /          \
     *  catalina   shared.
     */
   private void initClassLoaders() {
       try {
        //创建 common 类加载器
           commonLoader = createClassLoader("common", null);
           if( commonLoader == null ) {
               // no config file, default to this loader - we might be in
a 'single' env.
               commonLoader=this.getClass().getClassLoader();
           }
           //创建 catalina 类加载器,指定其父级别的加载器为 commonLoader.
           catalinaLoader = createClassLoader("server", commonLoader);
           //创建 sharedLoader 类加载器,指定其父级别的加载器为 commonLoader.
           sharedLoader = createClassLoader("shared", commonLoader);
       } catch (Throwable t) {
           log.error("Class loader creation threw exception", t);
           System.exit(1);
       }
   }
/**
     * 创建类加载器
     *
     * @param name
     * @param parent 指定上一级别的类加载器
     * @return
     * @throws Exception
     */
   private ClassLoader createClassLoader(String name, ClassLoader parent)
       throws Exception {
   //这里以 common 为例：从 catalina.properties 中获取 common.loader  类加载信息
```

```
//如:
//common.loader-${catalina.home}/lib,${catalina.home}/lib/*.jar
   String value = CatalinaProperties.getProperty(name + ".loader");
   // 如果没有任何信息，则返回父加载器
   if ((value == null) || (value.equals("")))
      return parent;

   ArrayList repositoryLocations = new ArrayList();
   ArrayList repositoryTypes = new ArrayList();
   int i;
   //以逗号分隔.
   StringTokenizer tokenizer = new StringTokenizer(value, ",");

   while (tokenizer.hasMoreElements()) {
      String repository = tokenizer.nextToken();
      // Local repository
      boolean replace = false;
      String before = repository;
      //是否含有"${catalina.home}"
      while ((i=repository.indexOf(CATALINA_HOME_TOKEN))>=0) {
         replace=true;
         if (i>0) {
           //替换成 tomcat 路径，替换后的形式如下：
c:/opensource/tomcat5/lib.
         repository = repository.substring(0,i) + getCatalinaHome()
            + repository.substring(i+CATALINA_HOME_TOKEN.length());
         } else {
            repository = getCatalinaHome()
               + repository.substring(CATALINA_HOME_TOKEN.length());
         }
      }
      //是否含有"${catalina.base}"
      while ((i=repository.indexOf(CATALINA_BASE_TOKEN))>=0) {
         replace=true;
         if (i>0) {
           //同上，替换
         repository = repository.substring(0,i) + getCatalinaBase()
            + repository.substring(i+CATALINA_BASE_TOKEN.length());
         } else {
            repository = getCatalinaBase()
               + repository.substring(CATALINA_BASE_TOKEN.length());
         }
      }
      if (replace && log.isDebugEnabled())
         log.debug("Expanded " + before + " to " + repository);

      // Check for a JAR URL repository
      try {
         URL url=new URL(repository);
         repositoryLocations.add(repository);
         repositoryTypes.add(ClassLoaderFactory.IS_URL);
         continue;
      } catch (MalformedURLException e) {
         // Ignore
```

```
            }


            if (repository.endsWith("*.jar")) {
               repository = repository.substring
                  (0, repository.length() - "*.jar".length());
               repositoryLocations.add(repository);
               repositoryTypes.add(ClassLoaderFactory.IS_GLOB);
            } else if (repository.endsWith(".jar")) {
               repositoryLocations.add(repository);
               repositoryTypes.add(ClassLoaderFactory.IS_JAR);
            } else {
               repositoryLocations.add(repository);
               repositoryTypes.add(ClassLoaderFactory.IS_DIR);
            }
         }



         String[] locations = (String[]) repositoryLocations.toArray(new
String[0]);
         Integer[] types = (Integer[]) repositoryTypes.toArray(new
Integer[0]);

         //创建类加载器
         ClassLoader classLoader = ClassLoaderFactory.createClassLoader
            (locations, types, parent);

         // Retrieving MBean server
         MBeanServer mBeanServer = null;
         if (MBeanServerFactory.findMBeanServer(null).size() > 0) {
            mBeanServer =
               (MBeanServer)
MBeanServerFactory.findMBeanServer(null).get(0);
         } else {
            mBeanServer = ManagementFactory.getPlatformMBeanServer();
         }
         // Register the server classloader
         ObjectName objectName =
            new ObjectName("Catalina:type=ServerClassLoader,name=" + name);
         mBeanServer.registerMBean(classLoader, objectName);
         return classLoader;
     }
```

到此为止，已经了解了 Tomcat 初始化类加载器的过程。可能有的读者觉得还是不太理解，下面是笔者总结的加载顺序。

(1) 准备要启动 Tomcat，调用 Bootstrap 的 main 方法。

(2) 在 Tomcat 启动之前，需要加载类，就需要类加载器。于是调用方法完成初始化工作 init()。

(3) init()方法开始工作后再去调用 initClassLoaders()方法。

(4) 发现需要初始化 3 个类型的类加载器，再调用 createClassLoader (name,parent)，通

过此方法告诉它要初始化哪种类型的。

(5) 通过 CatalinaProperties 类去联络 catalina.properties，获得哪种类加载器加载哪些类的信息。

(6) 完成初始化，并返回结果。

(7) Tomcat 加载其他资源(待续)，启动成功。

由此可见，Tomcat 加载器的实现清晰易懂，并且采用了官方推荐的“正统”的使用类加载器的方式。如果读者阅读完上面的案例后，能毫不费力地完全理解 Tomcat 设计团队这样布置加载器架构的用意，那说明您已经大致掌握了类加载器“主流”的使用方式，那么笔者不妨再提一个问题让各位思考一下：如果有 10 个 Web 应用程序都是用 Spring 来进行组织和管理的话，可以把 Spring 放到 Common 或 Shared 目录下让这些程序共享。Spring 要对用户程序的类进行管理，自然要能访问到用户程序的类，而用户的程序显然是放在 /WebApp/WEB-INF 目录中的。

13.2　OSGi 的类加载器架构

OSGi 是 Open Service Gateway Initiative 的缩写，是 OSGi 联盟(OSGi Alliance)制订的一个基于 Java 语言的动态模块化规范，这个规范最初由 Sun、IBM、爱立信等公司联合发起，目的是使服务提供商通过住宅网关为各种家用智能设备提供各种服务，后来这个规范在 Java 的其他技术领域也有相当不错的发展，现在已经成为 Java 世界中“事实上”的模块化标准，并且已经有了 Equinox、Felix 等成熟的实现。OSGi 在 Java 程序员中最著名的应用案例就是 Eclipse IDE，另外还有许多大型的软件平台和中间件服务器都基于或声明将会基于 OSGi 规范来实现，如 IBM Jazz 平台、GlassFish 服务器、Weblogic10.3 所使用的 mSA 架构等。

OSGi 中的每个模块(称为 Bundle)与普通的 Java 类库区别并不太大，两者一般都以 JAR 格式进行封装，并且内部存储的都是 Java Package 和 Class。但是一个 Bundle 可以声明它所依赖的 Java Package(通过 Import-Package 描述)，也可以声明它允许导出发布的 Java Package(通过 Export-Package 描述)。在 OSGi 里面，Bundle 之间的依赖关系从传统的上层模块依赖底层模块转变为平级模块之间的依赖(至少外观上是如此)，而且类库的可见性能得到了非常精确的控制，一个模块里只有被 Export 过的 Package 才可能被外界访问，其他的 Package 和 Class 将会被隐藏起来。除了更精确的模块划分和可见性控制外，引入 OSGi 的另外一个重要理由是，基于 OSGi 的程序很可能(只是很可能，并不是一定会)可以实现模块级的热插拔功能，当程序升级更新或调试除错时，可以只停用、重新安装，然后启用程序其中的一部分，这对企业级程序开发来说是一个非常有诱惑力的特性。

OSGi 之所以能有上述诱人的特点，要归功于它灵活的类加载器架构。OSGi 的 Bundle 类加载器之间只有规则，没有固定的委派关系。例如，某个 Bundle 声明了一个它依赖的 Package，如果有其他 Bundle 声明发布了这个 Package 后，那么对这个 Package 的所有类加载动作都会委派给发布它的 Bundle 类加载器去完成。不涉及某个具体的 Package 时，各个

Bundle 加载器都是平级的关系，只有具体使用到某个 Package 和 Class 的时候，才会根据 Package 导入导出定义来构造 Bundle 间的委派和依赖。

另外，一个 Bundle 类加载器为其他 Bundle 提供服务时，会根据 Export-Package 列表严格控制访问范围。如果一个类存在于 Bundle 的类库中但是没有被 Export，那么这个 Bundle 的类加载器能找到这个类，但不会提供给其他 Bundle 使用，而且 OSGi 平台也不会把其他 Bundle 的类加载请求分配给这个 Bundle 来处理。

我们可以举一个更具体一些的简单例子，假设存在 Bundle A、Bundle B 和 Bundle C 三个模块，并且这三个 Bundle 定义的依赖关系为：

- Bundle A：声明发布了 packageA，依赖了 java.*的包；
- Bundle B：声明依赖了 packageA 和 packageC，同时也依赖了 java.*的包；
- Bundle C：声明发布了 packageC，依赖了 packageA。

那么，这三个 Bundle 之间的类加载器及父类加载器之间的关系如图 13-4 所示。

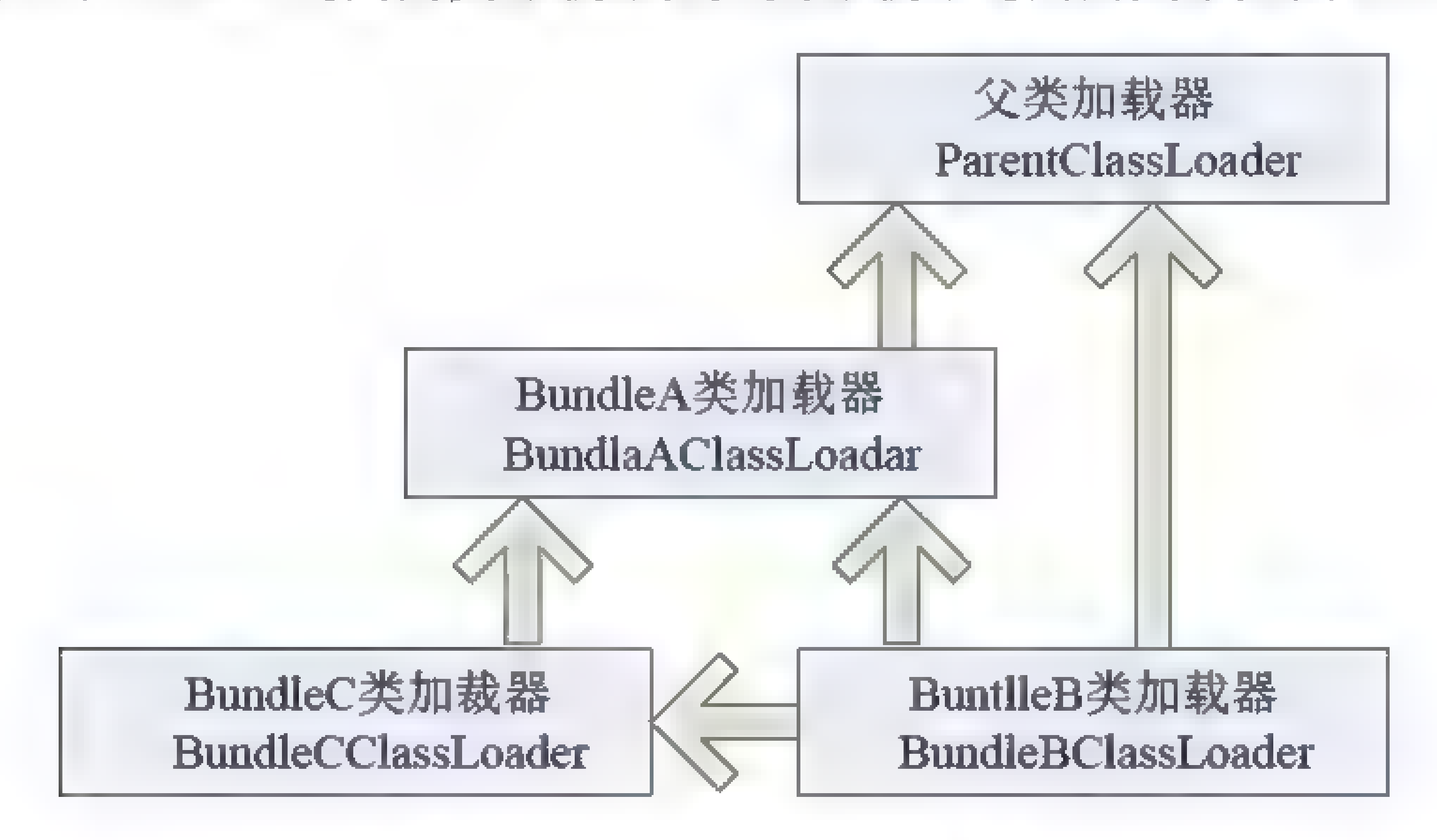


图 13-4 OSGi 的类加载器架构

由此可以看出，OSGi 类加载器之间不再是双亲委派模型的树形结构，而是进一步发展成 为一种运行时才能确定的网状结构。这种网状类加载器结构拥有更优秀的灵活性，同时也有许多隐患，由于各模块间依赖关系错综复杂，高并发量下容易发生死锁等问题。

由于没有牵涉到具体的 OSGi 实现，图中的类加载器都没有指明具体的加载器实现，只是一个体现了加载器间关系的概念模型，并且只是体现了 OSGi 中最简单的加载器委派关系。一般来说，在 OSGi 里，加载一个类可能发生的查找行为和委派关系会比图 13-4 中显示的复杂得多，类加载时可能进行的查找规则如下：

以 java.*开头的类，委派给父类加载器加载。

- 否则，委派列表名单内的类，委派给父类加载器加载。
- 否则，Import 列表中的类，委派给 Export 这个类的 Bundle 的类加载器加载。
- 否则，查找当前 Bundle 的 Classpath，使用自己的类加载器加载。
- 否则，查找是否在自己的 Fragment Bundle 中，如果是则委派给 Fragment Bundle 的类加载器加载。
- 否则，查找 Dynamic Import 列表的 Bundle，委派给对应 Bundle 的类加载器

加载。

- 否则，类查找失败。

从图 13-4 中还可以看出，在 OSGi 里面，加载器之间的关系不再是双亲委派模型的树形结构，而是已经进一步发展成了一种运行时才能确定的网状结构。这种网状的类加载器架构在带来更优秀的灵活性的同时，也可能会产生许多新的隐患。笔者曾经参与过将一个非 OSGi 的大型系统向 Equinox OSGi 平台迁移的项目，由于历史原因，代码模块之间的依赖关系错综复杂，勉强分离出各个模块的 Bundle 后，发现在高并发环境下经常出现死锁。我们很容易就找到了死锁的原因：如果出现了 Bundle A 依赖 Bundle B 的 Package B，而 Bundle B 又依赖了 Bundle A 的 Package A，这两个 Bundle 进行类加载时就很容易发生死锁。具体情况是当 Bundle A 加载 Package B 的类时，首先需要锁定当前类加载器的实例对象(java.lang.ClassLoader.loadClass()是一个 synchronized 方法)，然后把请求委派给 Bundle B 的加载器处理，但如果这时候 Bundle B 也正好想加载 Package A 的类，它也先锁定自己的加载器再去请求 Bundle A 的加载器处理，这样两个加载器都在等待对方处理自己的请求，而对方处理完之前自己又一直处于同步锁定的状态，因此它们就互相死锁，永远无法完成加载请求了。Equinox 的 Bug List 中也有关于这类问题的 Bug，并提供了一个以牺牲性能为代价的解决方案——用户可以启用 osgi.classloader.singleThreadLoads 参数来按单线程串行化的方式强制进行类加载动作。在 JDK 1.7 中，将会为非树状继承关系下的类加载器架构进行一次专门的升级，希望从底层避免这类死锁状况，这个动作也是为将来实现“官方的”模块化规范(与前文所提的 OSGi 是“事实上”的模块化规范对应)JSR-297、JSR-277 做准备。

总体来说，OSGi 描绘了一个很美好的模块化开发的目标，而且定义了实现这个目标所需要的各种服务，同时也有成熟框架对其提供实现支持。对于单个虚拟机下的应用，从开发初期就建立在 OSGi 上是一个很不错的选择，这样便于约束依赖。但并非所有的应用都适合采用 OSGi 作为基础架构，OSGi 在提供强大功能的同时，也引入了额外的复杂度，带来了线程死锁和内存泄漏的风险。

13.3　字节码生成技术

“字节码生成”并不是什么高深的技术，读者在看到“字节码生成”这个标题时也不必先去想诸如 JaVassist、CGLib 和 ASM 之类的字节码类库，因为 JDK 里面的 javac 命令就是字节码生成技术的“老祖宗”，并且 javac 也是一个由 Java 语言写成的程序，它的代码存放在 OpenjDK 的 jdk7/langtools/src/share/classes/com/sun/tools/javac 目录中。

要深入了解字节码生成，阅读 javac 的源码是个很好的途径，不过 javac 对于我们这个例子来说太过庞大了。在 Java 里面除了 javac 和字节码类库外，使用到字节码生成的例子还有很多，如 Web 服务器中的 JSP 编译器，编译时织入的 AOP 框架，还有很常用的动态代理技术，甚至在使用反射的时候虚拟机都有可能会在运行时生成字节码来提高执行速度。我们选择其中相对简单的动态代理来看看字节码生成技术是如何影响程序运作的。

其实 Java 编译程序将 Java 源程序翻译为 JVM 可执行代码?字节码。这一编译过程同

C/C++的编译有些不同。当 C 编译器编译生成一个对象的代码时，该代码是为在某一特定硬件平台运行而产生的。因此，在编译过程中，编译程序通过查表将所有对符号的引用转换为特定的内存偏移量，以保证程序运行。Java 编译器却不将对变量和方法的引用编译为数值引用，也不确定程序执行过程中的内存布局，而是将这些符号引用信息保留在字节码中，由解释器在运行过程中创立内存布局，然后再通过查表来确定一个方法所在的地址。这样就有效地保证了 Java 的可移植性和安全性。

运行 JVM 字节码的工作是由解释器来完成的。解释执行过程分三部进行：代码的装入、代码的校验和代码的执行。装入代码的工作由“类装载器”(Class Loader)完成。类装载器负责装入运行一个程序需要的所有代码，这也包括程序代码中的类所继承的类和被其调用的类。当类装载器装入一个类时，该类被放在自己的名字空间中。除了通过符号引用自己名字空间以外的类，类之间没有其他办法可以影响其他类。在本台计算机上的所有类都在同一地址空间内，而所有从外部引进的类，都有一个自己独立的名字空间。这使得本地类通过共享相同的名字空间获得较高的运行效率，同时又保证它们与从外部引进的类不会相互影响。当装入了运行程序需要的所有类后，解释器便可确定整个可执行程序的内存布局。解释器为符号引用同特定的地址空间建立对应关系及查询表。通过在这一阶段确定代码的内存布局，Java 很好地解决了由超类改变而使子类崩溃的问题，同时也防止了代码对地址的非法访问。

随后，被装入的代码由字节码校验器进行检查。校验器可发现操作数栈溢出，非法数据类型转化等多种错误。通过校验后，代码便开始执行了。

Java 字节码的执行有如下两种方式：

(1) 即时编译方式：解释器先将字节码编译成机器码，然后再执行该机器码。

(2) 解释执行方式：解释器通过每次解释并执行一小段代码来完成 Java 字节码程 序的所有操作。

通常采用的是第二种方法。由于 JVM 规格描述具有足够的灵活性，这使得将字节码翻译为机器代码的工作具有较高的效率。对于那些对运行速度要求较高的应用程序，解释器可将 Java 字节码即时编译为机器码，从而很好地保证了 Java 代码的可移植性和高性能。

13.4 动 态 代 理

所谓动态代理，就是在程序运行的时候，JVM 能动态的知道它要对哪个类的哪些方法进行代理，以此实现程序的可重用性，也减少了程序员的劳动量。代理是一种常用的设计模式，其目的就是为其他对象提供一个代理以控制对某个对象的访问。本节将简单介绍动态代理的基本知识。

13.4.1 代理模式

代理类负责为委托类预处理消息，过滤消息并转发消息，以及进行消息被委托类执行后的后续处理。例如图 13-5 所示的代理模式结构。

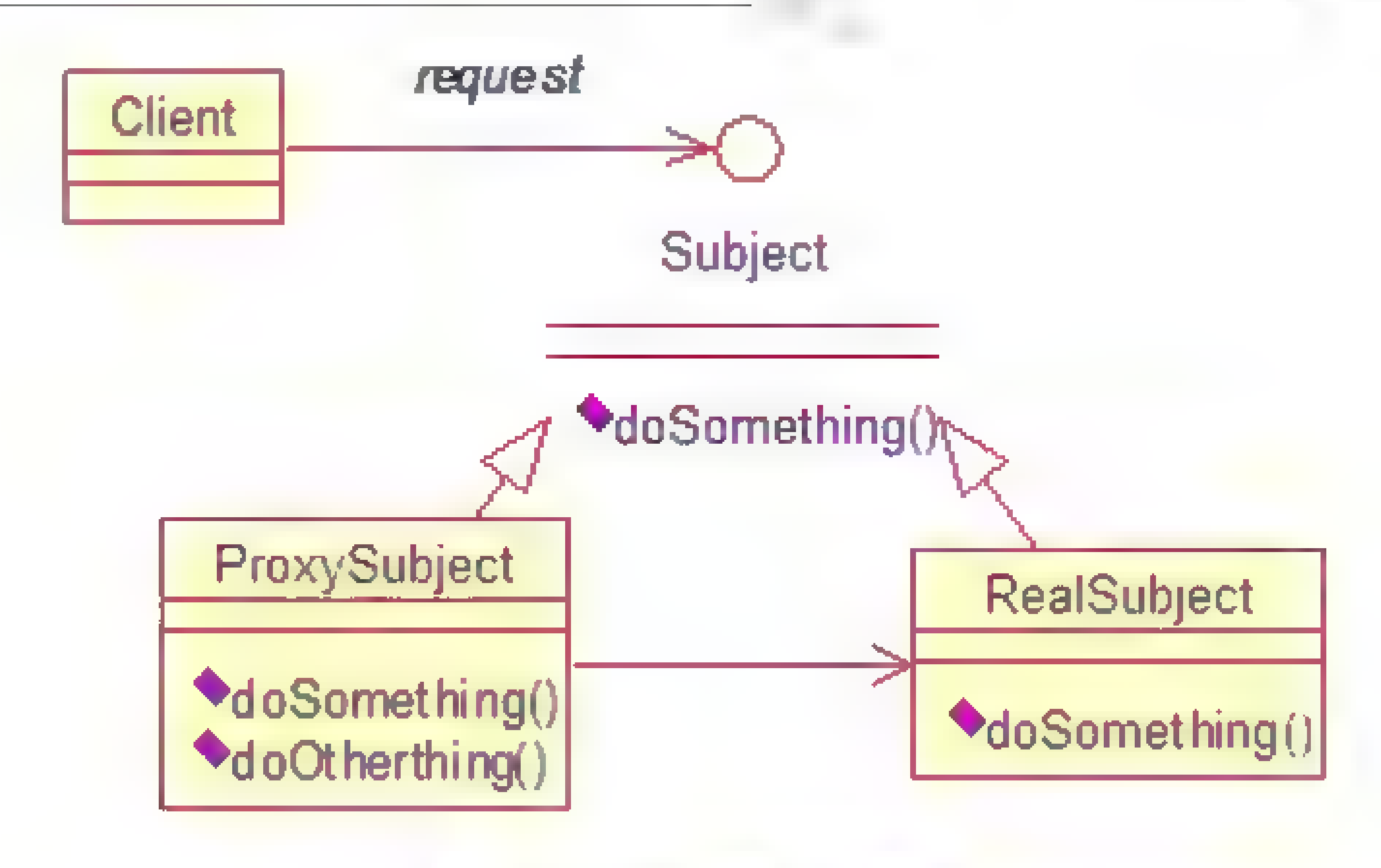


图 13-5　代理模式结构

为了保持行为的一致性，代理类和委托类通常会实现相同的接口，所以在访问者看来两者没有丝毫的区别。通过代理类这中间一层，能有效控制对委托类对象的直接 访问，也可以很好地隐藏和保护委托类对象，同时也为实施不同控制策略预留了空间，从而在设计上获得了更大的灵活性。Java 动态代理机制以巧妙的方式近乎完美地实践了代理模式的设计理念。

相信许多 Java 开发人员都使用过动态代理，即使没有直接使用过 java.lang.reflect.Proxy 或实现过 java.lang.re flect.InvocationHandler 接口，应该也用过 Spring 来做过 Bean 的组织管理。如果使用过 Spring，那大多数情况下就都用过动态代理，因为如果 Bean 是面向接口编程，那么在 Spring 内部则都是通过动态代理的方式来对 Bean 进行增强的。

动态代理中所谓的"动态"，是针对使用 Java 代码实际编写了代理类的"静态"代理而言的，它的优势不在于省去了编写代理类那一点工作量，而是实现了可以在原始类和接口还未知的时候，就确定代理类的代理行为，当代理类与原始类脱离直接联系后，就可以很灵活地重用于不同的应用场景之中。

通过 JDK 的动态代理特性，可以为任意 Java 对象创建代理对象，对于具体使用来说，这个特性是通过 Java Reflection API 来完成的。在 Proxy 的调用过程中，如果客户(Client)调用 Proxy 的 request 方法，会在调用目标对象的 request 方法的前后调用一系列的处理，而这一系列的处理相对于目标对象来说是透明的，目标对象对这些处理可以毫不知情，这就是 Proxy 模式。

13.4.2　相关的类和接口

要了解 Java 动态代理的机制，首先需要了解以下相关的类或接口。

(1) java.lang.reflect.Proxy：这是 Java 动态代理机制的主类，它提供了一组静态方法来为一组接口动态地生成代理类及其对象。例如下面的代码演示了 Proxy 的静态方法：

```
// 方法 1：该方法用于获取指定代理对象所关联的调用处理器
static InvocationHandler getInvocationHandler(Object proxy)
// 方法 2：该方法用于获取关联于指定类装载器和一组接口的动态代理类的类对象
static Class getProxyClass(ClassLoader loader, Class[] interfaces)
// 方法 3：该方法用于判断指定类对象是否是一个动态代理类
```

```
static boolean isProxyClass(Class cl)
// 方法 4：该方法用于为指定类装载器、一组接口及调用处理器生成动态代理类实例
static Object newProxyInstance(ClassLoader loader, Class[] interfaces,
    InvocationHandler h)
```

(2) java.lang.reflect.InvocationHandler：这是调用处理器接口，它自定义了一个 invoke 方法，用于集中处理在动态代理类对象上的方法调用，通常在该方法中实现对委托类的代理访问。例如下面的代码演示 InvocationHandler 的核心方法：

```
// 该方法负责集中处理动态代理类上的所有方法调用。第一个参数既是代理类实例，第二个参数是被调
用的方法对象
// 第三个方法是调用参数。调用处理器根据这三个参数进行预处理或分派到委托类实例上发射执行
Object invoke(Object proxy, Method method, Object[] args)
```

每次生成动态代理类对象时都需要指定一个实现了该接口的调用处理器对象。

(3) java.lang.ClassLoader：这是类装载器类，负责将类的字节码装载到 Java 虚拟机(JVM)中并为其定义类对象，然后该类才能被使用。Proxy 静态方法生成动态代理类同样需要通过类装载器来进行装载才能使用，它与普通类的唯一区别就是其字节码是由 JVM 在运行时动态生成的而非预存在于任何一个 .class 文件中。

每次生成动态代理类对象时都需要指定一个类装载器对象。

13.4.3　代理机制及其特点

首先让我们来了解一下如何使用 Java 动态代理，具体有以下四步。

(1) 通过实现 InvocationHandler 接口创建自己的调用处理器；

(2) 通过为 Proxy 类指定 ClassLoader 对象和一组 interface 来创建动态代理类；

(3) 通过反射机制获得动态代理类的构造函数，其唯一参数类型是调用处理器接口类型；

(4) 通过构造函数创建动态代理类实例，构造时调用处理器对象作为参数被传入。

如下代码演示了动态代理对象创建过程：

```
// InvocationHandlerImpl 实现了 InvocationHandler 接口，并能实现方法调用从代理类到委托
类的分派转发
// 其内部通常包含指向委托类实例的引用，用于真正执行分派转发过来的方法调用
InvocationHandler handler = new InvocationHandlerImpl(..);
// 通过 Proxy 为包括 Interface 接口在内的一组接口动态创建代理类的类对象
Class clazz = Proxy.getProxyClass(classLoader, new Class[]
{ Interface.class, ... });
// 通过反射从生成的类对象获得构造函数对象
Constructor constructor = clazz.getConstructor(new Class[]
{ InvocationHandler.class });
// 通过构造函数对象创建动态代理类实例
Interface Proxy = (Interface)constructor.newInstance(new Object[] { handler });
```

实际使用过程更加简单，因为 Proxy 的静态方法 newProxyInstance 已经为我们封装了步骤(2)到步骤(4)的过程，所以简化后的过程如下：

```
// InvocationHandlerImpl 实现了 InvocationHandler 接口，并能实现方法调用从代理
类到委托类的分派转发
```

```
InvocationHandler handler = new InvocationHandlerImpl(..);
// 通过 Proxy 直接创建动态代理类实例
Interface proxy = (Interface)Proxy.newProxyInstance( classLoader,
     new Class[] { Interface.class },
     handler );
```

接下来让我们来了解一下 Java 动态代理机制的一些特点。

首先是动态生成的代理类本身的一些特点。

(1) 包：如果所代理的接口都是 public 的，那么它将被定义在顶层包(即包路径为空)，如果所代理的接口中有非 public 的接口(因为接口不能被定义为 protect 或 private，所以除 public 之外就是默认的 package 访问级别)，那么它将被定义在该接口所在包(假设代理了 com.ibm.developerworks 包中的某非 public 接口 A，那么新生成的代理类所在的包就是 com.ibm.developerworks)，这样设计的目的是为了最大程度的保证动态代理类不会因为包管理的问题而无法被成功定义并访问；

(2) 类修饰符：该代理类具有 final 和 public 修饰符，意味着它可以被所有的类访问，但是不能被再度继承；

(3) 类名：格式是“$ProxyN”，其中 N 是一个逐一递增的阿拉伯数字，代表 Proxy 类第 N 次生成的动态代理类，值得注意的一点是，并不是每次调用 Proxy 的静态方法创建动态代理类都会使得 N 值增加，原因是如果对同一组接口(包括接口排列的顺序相同)试图重复创建动态代理类，它会很聪明地返回先前已经创建好的代理类的类对象，而不会再尝试去创建一个全新的代理类，这样可以节省不必要的代码重复生成，提高了代理类的创建效率。

(4) 类继承关系：该类的继承关系如图 13-6 所示。

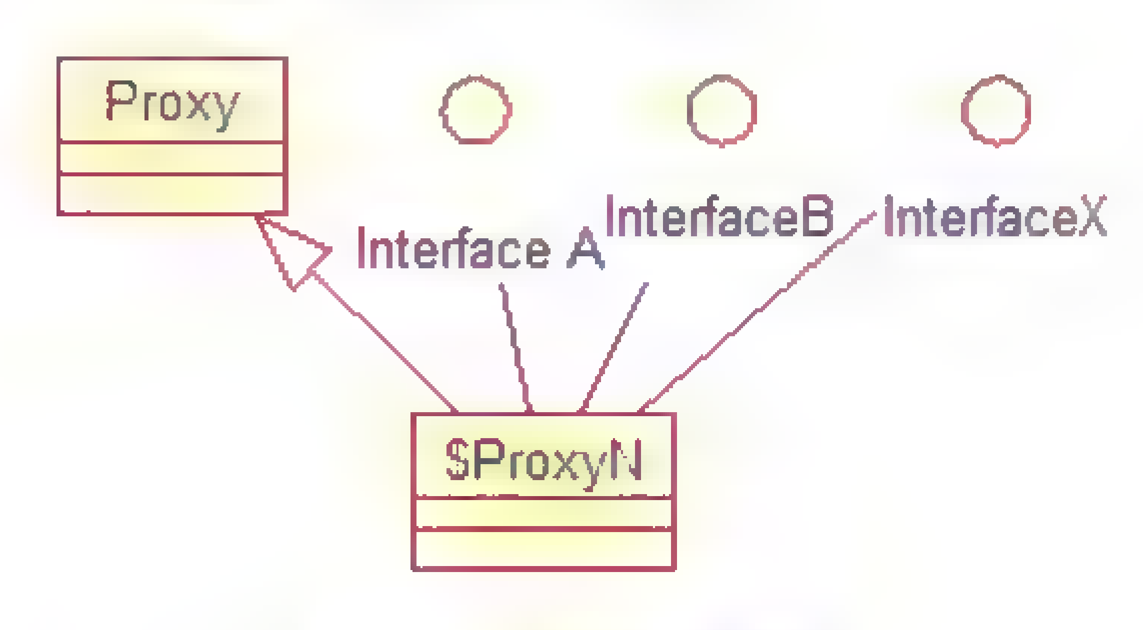


图 13-6 类继承关系图

由图 13-6 可见，类 Proxy 是它的父类，这个规则适用于所有由 Proxy 创建的动态代理类。而且该类还实现了其所代理的一组接口，这就是为什么它能够被安全地类型转换到其所代理的某接口的根本原因。

接下来让我们了解一下代理类实例的一些特点。每个实例都会关联一个调用处理器对象，可以通过 Proxy 提供的静态方法 getInvocationHandler 去获得代理类实例的调用处理器对象。在代理类实例上调用其代理的接口中所声明的方法时，这些方法最终都会由调用处理器的 invoke 方法执行。此外，值得注意的是，代理类的根类 java.lang.Object 中有三个方法也同样会被分派到调用处理器的 invoke 方法执行，它们是 hashCode、equals 和 toString，可能的原因有：一是因为这些方法为 public 且非 final 类型，能够被代理类覆

盖；二是因为这些方法往往呈现出一个类的某种特征属性，具有一定的区分度，所以为了保证代理类与委托类对外的一致性，这三个方法也应该被分派到委托类执行。当代理的一组接口有重复声明的方法且该方法被调用时，代理类总是从排在最前面的接口中获取方法对象并分派给调用处理器，而无论代理类实例是否正在以该接口(或继承于该接口的某子接口)的形式被外部引用，因为在代理类内部无法区分其当前的被引用类型。

13.4.4 应用动态代理

接着来了解一下被代理的一组接口有哪些特点。首先，要注意不能有重复的接口，以避免动态代理类代码生成时的编译错误。其次，这些接口对于类装载器必须可见，否则类装载器将无法链接它们，将会导致类定义失败。再次，需被代理的所有非 public 的接口必须在同一个包中，否则代理类生成也会失败。最后，接口的数目不能超过 65535，这是 JVM 设定的限制。

最后再来了解一下异常处理方面的特点。从调用处理器接口声明的方法中可以看到理论上它能够抛出任何类型的异常，因为所有的异常都继承于 Throwable 接口，但事实是否如此呢？答案是否定的，原因是我们必须遵守一个继承原则：即子类覆盖父类或实现父接口的方法时，抛出的异常必须在原方法支持的异常列表之内。所以虽然调用处理器理论上讲能够，但实际上往往受限制，除非父接口中的方法支持抛 Throwable 异常。那么如果在 invoke 方法中的确产生了接口方法声明中不支持的异常，那将如何呢？放心，Java 动态代理类已经为我们设计好了解决方法：它将会抛出 UndeclaredThrowableException 异常。此异常是一个 RuntimeException 类型，所以不会引起编译错误。通过该异常的 getCause 方法，还可以获得原来那个不受支持的异常对象，以便于错误诊断。

在 JDK 中已经实现了这个 Proxy 模式，在基于 Java 虚拟机设计应用程序时，只需要直接使用这个特性就可以了。具体来说，可以在 Java 的 reflection 包中看到 Proxy 对象，这个对象生成后，所起的作用就类似于 Proxy 模式中的 Proxy 对象。在使用时，还需要为代理对象(Proxy)设计一个回调方法，这个回调方法起到的作用是，在其中加入了作为代理需要额外处理的动作，或者说，在这个方法中，所谓额外动作，可以参考 Proxy 模式中的 preOperation()和 postOperation()方法。这个回调方法，如果在 JDK 中实现，需要实现下面的 InvocationHandler 接口：

```
public interface InvocationHandler {
public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable;
}
```

在这个接口方法中，只声明了一个 invoke 方法，这个 invoke 方法的第一个参数是代理对象实例，第二个参数是 Method 方法对象，代表的是当前 Proxy 被调用的方法，最后一个参数是被调用的方法中的参数。通过这些信息，在 invoke 方法实现中，已经可以了解 Proxy 对象的调用背景了。至于怎样让 invoke 方法和 Proxy 挂上钩，熟悉 Proxy 用法的读者都知道，只要在实现通过调用 Proxy.newIntance 方法生成具体 Proxy 对象时把 InvocationHandler 设置到参数里面就可以了，剩下的由 Java 虚拟机来完成。

如果读者研究过 JDK 源代码，则会发现在 Proxy 的 sun 实现中调用了 sun.misc.ProxyGenerator 类的 generateProxyClass(proxyName, interfaces) 方法，其返回值为 byte[] 和 class 文件的内存类型一致。请看下面的演示代码：

```
public class  ProxyClassFile{
      public static void main(String[] args){
            String proxyName = "TempProxy";
       TempImpl t = new TempImpl("proxy");
            Class[] interfaces =t.getClass().getInterfaces();
            byte[] proxyClassFile = ProxyGenerator.generateProxyClass(
                proxyName, interfaces);
       File f = new File("classes/TempProxy.class");
       try {
          FileOutputStream fos = new FileOutputStream(f);
          fos.write(proxyClassFile);
          fos.flush();
          fos.close();
       } catch (FileNotFoundException e) {
          e.printStackTrace();  //To change body of catch statement use
File | Settings | File Templates.
       } catch (IOException e) {
          e.printStackTrace();  //To change body of catch statement use
File | Settings | File Templates.
       }
      }
}
```

运行该类，到 Class 文件夹下，利用反编译技术，发现原来其采用了代码生产技术：

```
public interface Temp{
      public void Talk();
      public void Run();
}
import java.lang.reflect.*;
public final class TempProxy extends Proxy
   implements Temp{
   private static Method m4;
   private static Method m2;
   private static Method m0;
   private static Method m3;
   private static Method m1;
   public TempProxy(InvocationHandler invocationhandler)   {
      super(invocationhandler);
   }
   public final void Run()    {
      try {
         h.invoke(this, m4, null);
         return;
      }
      catch(Error  ex) { }
      catch(Throwable throwable)  {
         throw new UndeclaredThrowableException(throwable);
      }
   }
```

```
    public final String toString(){
        try{
            return (String)h.invoke(this, m2, null);
        }
        catch(Error  ex) { }
        catch(Throwable throwable)  {
            throw new UndeclaredThrowableException(throwable);
        }
        return "";
    }
    public final int hashCode() {
        try {
            return ((Integer)h.invoke(this, m0, null)).intValue();
        }
        catch(Error  ex) { }
        catch(Throwable throwable){
            throw new UndeclaredThrowableException(throwable);
        }
        return 123;
    }
    public final void Talk(){
        try{
            h.invoke(this, m3, null);
            return;
        }
        catch(Error  ex) { }
        catch(Throwable throwable) {
            throw new UndeclaredThrowableException(throwable);
        }
    }
    public final boolean equals(Object obj) {
        try  {
            return ((Boolean)h.invoke(this, m1, new Object[] {
                obj
            })).booleanValue();
        }
        catch(Error  ex) { }
        catch(Throwable throwable) {
            throw new UndeclaredThrowableException(throwable);
        }
        return false;
    }
    static{
        try{
     m4 = Class.forName("Temp").getMethod("Run", new Class[0]);
     m2 = Class.forName("java.lang.Object").getMethod("toString", new Class[0]);
     m0 = Class.forName("java.lang.Object").getMethod("hashCode", new Class[0]);
     m3 = Class.forName("Temp").getMethod("Talk", new Class[0]);
     m1 = Class.forName("java.lang.Object").getMethod("equals", new
Class[] {
                Class.forName("java.lang.Object")
            });
        }
        catch(NoSuchMethodException nosuchmethodexception) {
```

```
            throw new
NoSuchMethodError(nosuchmethodexception.getMessage());
        }
        catch(ClassNotFoundException classnotfoundexception) {
            throw new
NoClassDefFoundError(classnotfoundexception.getMessage());
        }
    }
}
```

一个动态代理必须要有的 4 个类：一个接口、一个实现了该接口的类、一个实现 InvocationHandler 接口的类和写一个有 main 方法的测试类。这里需要重点说的是，实现了 InvocationHandler 接口的类，在 InvocationHandler 接口里有且只有一个方法：invoke()，真正的代理是由此方法来实现的。它需要三个参数：要代理的类、要代理的方法，以及方法的参数。所以实现了 InvoactionHandler 接口的类必须实现 invoke()这个方法，在这个方法里我们可以利用反射的原理调用指定类的具有指定参数的方法，也就是实现代理。

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.util.logging.Logger;
public class LogHandler implements InvocationHandler
{
 private Logger log=Logger.getLogger(this.getClass().getName());
 private Object delegate;
 public LogHandler(Object delegate)
 {
     this.delegate=delegate;
 }
 public Object invoke(Object proxy,Method method,Object[] args) throws
Throwable
 {
    Object obj=null;
    try
    {
          log.info("method stats...."+method);
          obj=method.invoke(delegate,args);
          log.info("method end...."+method);
    }
    catch(Exception e)
    {
          log.info("Exception happends....");
    }
    return obj;
 }
}
```

另外一个需要说的就是有 main()方法的测试类，在这个 main()方法里我们需要用真实类创建一个对象，并以此对象为参数实例化一个 InvocationHandler(参考 jdk1.5 中的写法 InvocationHandler handler = new MyInvocationHandler(...))，最核心的一步就是用 Proxy 类得到一个接口类的代理类，我们可以参考 jdk1.5 中的写法：

```
Foo f = (Foo) Proxy.newProxyInstance(Foo.class.getClassLoader(),new
```

```
Class[] {
 Foo.class
},
handler);
```

其中 Foo 是原始类，即被代理的类，用这个代理类来调用原始类中的方法，它是通过 invoke()方法实现的，至此我们就完成了动态代理。测试类程序清单如下：

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Proxy;
public class BusinessObject
{
     public static void main(String args[])
     {
           BussObj businessImp=new BussObj();
           InvocationHandler handler=new LogHandler(businessImp);
           Business
business=(Business)Proxy.newProxyInstance(Business.class.getClassLoader(
),businessImp.getClass().getInterfaces(),handler);
           business.getString();
   }
  }
```

其中 Business 是接口，BussObj 类是实现了 Business 接口的类，这两个类中的代码需要读者自己去写。上述测试程序运行结果如图 13-7 所示。

```
C:\itcast\day7动态代理>java BusinessObject
2006-9-3 17:21:57 LogHandler invoke
信息: method stats....public abstract void Business.getString()
0,1,2,3
2006-9-3 17:21:58 LogHandler invoke
信息: method end....public abstract void Business.getString()

C:\itcast\day7动态代理>
```

图 13-7　执行效果

在使用代理时，首先记住如下 Proxy 的几个重要的静态变量：

```
// 映射表：用于维护类装载器对象到其对应的代理类缓存
private static Map loaderToCache = new WeakHashMap();
// 标记：用于标记一个动态代理类正在被创建中
private static Object pendingGenerationMarker = new Object();
// 同步表：记录已经被创建的动态代理类类型，主要被方法 isProxyClass 进行相关的判断
private static Map proxyClasses = Collections.synchronizedMap(new
WeakHashMap());
// 关联的调用处理器引用
protected InvocationHandler h;
```

然后看一下 Proxy 的构造方法：

```
// 由于 Proxy 内部从不直接调用构造函数，所以 private 类型意味着禁止任何调用
private Proxy() {}
// 由于 Proxy 内部从不直接调用构造函数，所以 protected 意味着只有子类可以调用
```

```
protected Proxy(InvocationHandler h) {this.h - h;}
接着可以快速浏览 newProxyInstance 方法:
public static Object newProxyInstance(ClassLoader loader,
          Class<?>[] interfaces,
          InvocationHandler h)
          throws IllegalArgumentException {

    // 检查 h 不为空，否则抛异常
    if (h == null) {
       throw new NullPointerException();
    }

    // 获得与制定类装载器和一组接口相关的代理类类型对象
    Class cl = getProxyClass(loader, interfaces);
    // 通过反射获取构造函数对象并生成代理类实例
    try {
       Constructor cons = cl.getConstructor(constructorParams);
       return (Object) cons.newInstance(new Object[] { h });
    } catch (NoSuchMethodException e) { throw new
InternalError(e.toString());
    } catch (IllegalAccessException e) { throw new
InternalError(e.toString());
    } catch (InstantiationException e) { throw new
InternalError(e.toString());
    } catch (InvocationTargetException e) { throw new
InternalError(e.toString());
    }
}
```

由此可见，动态代理真正的关键是在 getProxyClass 方法，该方法负责为一组接口动态地生成代理类类型对象。在该方法内部，您将能看到 Proxy 内的各路英雄(静态变量)悉数登场。有点迫不及待了么？那就让我们一起走进 Proxy 最最神秘的殿堂去欣赏一番吧。该方法总共可以分为四步。

(1) 对这组接口进行一定程度的安全检查，包括检查接口类对象是否对类装载器可见并且与类装载器所能识别的接口类对象是完全相同的，还会检查确保是 interface 类型而不是 class 类型。这个步骤通过一个循环来完成，检查通过后将会得到一个包含所有接口名称的字符串数组，记为 String[] interfaceNames。总体上这部分实现比较直观，所以略去大部分代码，仅保留如何判断某类或接口是否对特定类装载器可见的相关代码。例如在下面的代码中，通过方法 Class.forName 判断了接口的可见性。

```
    try {
       // 指定接口名字、类装载器对象，同时制定 initializeBoolean 为 false 表示无
需初始化类
       // 如果方法返回正常这表示可见，否则会抛出 ClassNotFoundException 异常表示
不可见
       interfaceClass = Class.forName(interfaceName, false, loader);
    } catch (ClassNotFoundException e) {
    }
```

(2) 从 loaderToCache 映射表中获取以类装载器对象为关键字所对应的缓存表，如果

不存在就创建一个新的缓存表并更新到 loaderToCache。缓存表是一个 HashMap 实例，正常情况下它将存放键值对(接口名字列表，动态生成的代理类的类对象引用)。当代理类正在被创建时它会临时保存(接口名字列表，pendingGenerationMarker)。标记 pendingGenerationMarke 的作用是通知后续的同类请求(接口数组相同且组内接口排列顺序也相同)代理类正在被创建，请保持等待直至创建完成。下面的代码演示了缓存表的使用。

```
do {
    // 以接口名字列表作为关键字获得对应 cache 值
    Object value = cache.get(key);
    if (value instanceof Reference) {
        proxyClass = (Class) ((Reference) value).get();
    }
    if (proxyClass != null) {
        // 如果已经创建，直接返回
        return proxyClass;
    } else if (value == pendingGenerationMarker) {
        // 代理类正在被创建，保持等待
        try {
            cache.wait();
        } catch (InterruptedException e) {
        }
        // 等待被唤醒，继续循环并通过二次检查以确保创建完成，否则重新等待
        continue;
    } else {
        // 标记代理类正在被创建
        cache.put(key, pendingGenerationMarker);
        // break 跳出循环已进入创建过程
        break;
} while (true);
```

(3) 动态创建代理类的类对象。首先是确定代理类所在的包，其原则如前所述，如果都为 public 接口，则包名为空字符串表示顶层包；如果所有非 public 接口都在同一个包，则包名与这些接口的包名相同；如果有多个非 public 接口且不同包，则抛异常终止代理类的生成。确定了包后，就开始生成代理类的类名，同样如前所述按格式“$ProxyN”生成。例如下面的代码动态生成了代理类。

```
    // 动态地生成代理类的字节码数组
    byte[] proxyClassFile = ProxyGenerator.generateProxyClass( proxyName,
interfaces);
    try {
        // 动态地定义新生成的代理类
        proxyClass = defineClass0(loader, proxyName, proxyClassFile, 0,
            proxyClassFile.length);
    } catch (ClassFormatError e) {
        throw new IllegalArgumentException(e.toString());
    }
    // 把生成的代理类的类对象记录进 proxyClasses 表
    proxyClasses.put(proxyClass, null);
```

由此可见，所有代码生成的工作都由神秘的 ProxyGenerator 完成了，当你尝试去探索这个类时，你所能获得的信息仅仅是它位于并未公开的 sun.misc 包，有若干常量、变量

和方法以完成这个神奇的代码生成的过程，但是 sun 并没有提供源代码以供研读。至于动态类的定义，则由 Proxy 的 native 静态方法 defineClass0 执行。

(4) 代码生成过程进入结尾部分，根据结果更新缓存表，如果成功则将代理类的类对象引用更新进缓存表，否则清除缓存表中对应的关键值，最后唤醒所有可能的正在等待的线程。

经过以上四步后，所有的代理类生成细节都已介绍完毕，剩下的静态方法如 getInvocationHandler 和 isProxyClass 就显得如此的直观，只需通过查询相关变量就可以完成，所以对其的代码分析就省略了。

分析了类 Proxy 的源代码之后，相信在读者的脑海中会对 Java 动态代理机制形成一个更加清晰的理解，但是当探索之旅在 sun.misc.ProxyGenerator 类处戛然而止，所有的神秘都汇聚于此时，相信不少读者也会对这个 ProxyGenerator 类产生有类似的疑惑：它到底做了什么呢？它是如何生成动态代理类的代码的呢？在此也无法给出确切的答案。还是让我们带着这些疑惑，一起开始探索之旅吧。

事物往往不像其看起来的复杂，需要的是我们能够化繁为简，这样也许就能有更多拨云见日的机会。抛开所有想象中的未知而复杂的神秘因素，如果让我们用最简单的方法去实现一个代理类，唯一的要求是同样结合调用处理器实施方法的分派转发，您的第一反应将是什么呢？"听起来似乎并不是很复杂"的确，掐指算算所涉及的工作无非包括几个反射调用，以及对原始类型数据的装箱或拆箱过程，其他的似乎都已经水到渠成。非常的好，让我们整理一下思绪，一起来完成一次完整的推演过程吧。

例如通过下面的代码，演示代理类中的方法调用的分派转发的推演实现过程。

```
// 假设需代理接口 Simulator
public interface Simulator {
    short simulate(int arg1, long arg2, String arg3) throws ExceptionA,
ExceptionB;
}
// 假设代理类为 SimulatorProxy, 其类声明将如下
final public class SimulatorProxy implements Simulator {
    // 调用处理器对象的引用
    protected InvocationHandler handler;
    // 以调用处理器为参数的构造函数
    public SimulatorProxy(InvocationHandler handler){
        this.handler = handler;
    }
    // 实现接口方法 simulate
    public short simulate(int arg1, long arg2, String arg3)
        throws ExceptionA, ExceptionB {
        // 第一步是获取 simulate 方法的 Method 对象
        java.lang.reflect.Method method = null;
        try{
            method = Simulator.class.getMethod(
                "simulate",
                new Class[] {int.class, long.class, String.class} );
        } catch(Exception e) {
            // 异常处理 1(略)
        }
```

```
        // 第二步是调用 handler 的 invoke 方法分派转发方法调用
        Object r = null;
        try {
            r = handler.invoke(this,
                method,
                // 对于原始类型参数需要进行装箱操作
                new Object[] {new Integer(arg1), new Long(arg2), arg3});
        }catch(Throwable e) {
            // 异常处理 2(略)
        }
        // 第三步是返回结果(返回类型是原始类型则需要进行拆箱操作)
        return ((Short)r).shortValue();
    }
}
```

模拟推演为了突出通用逻辑所以更多地关注正常流程，而淡化了错误处理，但在实际中错误处理同样非常重要。从以上的推演中我们可以得出一个非常通用的结构化流程：第一步从代理接口获取被调用的方法对象，第二步分派方法到调用处理器执行，第三步返回结果。在这之中，所有的信息都是可以已知的，比如接口名、方法名、参数类型、返回类型以及所需的装箱和拆箱操作，那么既然我们手工编写是如此，那又有什么理由不相信ProxyGenerator 不会做类似的实现呢？至少这是一种比较可能的实现。

接下来让我们把注意力重新回到先前被淡化的错误处理上来。在异常处理 1 处，由于我们有理由确保所有的信息如接口名、方法名和参数类型都准确无误，所以这部分异常发生的概率基本为零，所以基本可以忽略。而异常处理 2 处，我们需要思考得更多一些。回想一下，接口方法可能声明支持一个异常列表，而调用处理器 invoke 方法又可能抛出与接口方法不支持的异常，再回想一下先前提及的 Java 动态代理的关于异常处理的特点，对于不支持的异常，必须抛 UndeclaredThrowableException 运行时异常。所以通过再次推演，我们可以得出一个更加清晰的异常处理 2 的情况，下面是细化的异常处理 2 的演示代码。

```
Object r = null;
try {
    r = handler.invoke(this,
        method,
        new Object[] {new Integer(arg1), new Long(arg2), arg3});
} catch( ExceptionA e) {
    // 接口方法支持 ExceptionA，可以抛出
    throw e;
} catch( ExceptionB e ) {
    // 接口方法支持 ExceptionB，可以抛出
    throw e;
} catch(Throwable e) {
    // 其他不支持的异常，一律抛 UndeclaredThrowableException
    throw new UndeclaredThrowableException(e);
}
```

这样我们就完成了对动态代理类的推演实现。推演实现遵循了一个相对固定的模式，可以适用于任意定义的任何接口，而且代码生成所需的信息都是可知的，那么有理由相信

即使是机器自动编写的代码也有可能延续这样的风格，至少可以保证这是可行的。

注意：Proxy 已经设计得非常优美了，但是还是有一点点小小的遗憾，那就是它始终无法摆脱仅支持 interface 代理的桎梏，因为它的设计注定了这个遗憾。在动态生成的代理类的继承关系中，它们已经注定有一个共同的父类叫 Proxy。Java 的继承机制注定了这些动态代理类们无法实现对 Class 的动态代理，原因是多继承在 Java 中本质上就行不通。有很多条理由，人们可以否定对 Class 代理的必要性，但是同样有一些理由，相信支持 Class 动态代理会更美好。接口和类的划分，本就不是很明显，只是到了 Java 中才变得如此的细化。如果只从方法的声明及是否被定义来考量，有一种两者的混合体，它的名字叫抽象类。实现对抽象类的动态代理，相信也有其内在的价值。此外，还有一些历史遗留的类，它们将因为没有实现任何接口而从此与动态代理永世无缘。如此种种，不得不说是一个小小的遗憾。

第 14 章

编译优化

优化是指为了更加优秀，目的是“去其糟粕，取其精华”，是一项使某人/某物变得更优秀的方法/技术。从计算机程序出现的第一天起，对效率的追逐就是程序天生的坚定信仰，这个过程犹如一场没有终点、永不停歇的 F1 方程式赛车，程序员是车手，技术平台则是在赛道上飞驰的赛车。本章将详细讲解 JVM 在编译时期的优化技术方案。

14.1 Java 的编译过程

Java 编译程序能够将 Java 源程序编译成 JVM 可执行代码——Java 字节码。Java 在编译过程中一般会按照以下过程进行。

(1) JDK 根据编译参数 encoding 确定源代码字符集。如果不指定该参数，系统会根据操作系统的 file.encoding 参数来获取操作系统编码格式，国内的 Windows 通常都是 GBK。

(2) JDK 根据上面的字符集信息，将源文件编译成 JAVA 内部的 unicode 模式，并将编译后的内容保存到内存中。

(3) JDK 将内存保存完好的内存信息写入.class 文件，生成最后的二进制文件。

很多人会发现自己在 IDE 中配置了源文件字符集为 UTF-8(与操作系统默认字符集不同)后，再直接运行 javac 就会出现错误，这就是因为不加 encoding 参数的编译过程中会默认使用系统的字符集的。在 Windows 中默认为 GBK。IDE 中进行编译时，IDE 会在编译参数中增加该参数。

如果使用 Ant 来进行编译活动，那么需要确认源代码的字符集，然后再相应的 Ant 编译任务中，增加 encoding 参数就意味着，如果之前的项目采用的是系统默认的字符集(GBK)来编辑的源代码，那么如果改用 UTF-8，就意味需要重新修改 Ant 脚本，重新打包我们的产品。

Java 编译器却不将对变量和方法的引用编译为数值引用，也不确定程序执行过程中的内存布局，而是将些符号引用信息保留在字节码中，由解释器在运行过程中创立内存布局，然后再通过查表来确定一个方法所在的地址，这样就有效地保证了 Java 的可移植性和安全性。

运行 JVM 字符码的工作是由解释器来完成的。解释执行过程分三步进行：代码的装入、代码的校验、和代码的执行。

装入代码的工作由“类装载器 classloader”完成。类装载器负责装入运行一个程序需要的所有代码，这也包括程序代码中的类所继承的类和被调用的类。当类装载器装入一个类时，该类被放在自己的名字空间中。除了通过符号引用自己名字空间以外的类，类之间没有其他办法可以影响其他类。在本台计算机的所有类都在同一地址空间中，而所有从外部引进的类，都有一个自己独立的名字空间。这使得本地类通过共享相同的名字空间获得较高的运行效率，同时又保证它们与从外部引进的类不会相互影响。

当装入了运行程序需要的所有类后，解释器便可确定整个可执行程序的内存布局。解释器为符号引用与特定的地址空间建立对应关系及查询表。通过在这一阶段确定代码的内布局，Java 很好地解决了由超类改变而使子类崩溃的问题，同时也防止了代码的非法访问。

随后，被装入的代码由字节码校验器进行检查。校验器可以发现操作数栈溢出、非法数据类型转化等多种错误。通过校验后，代码便开始执行了。

有如下两种执行 Java 字节码的方式：

(1) 即时编译方式：解释器先将字节编译成机器码，然后再执行该机器码。

(2) 解释执行方式：解释器通过每次解释并执行一小段代码来完成 Java 字节码程序的所有操作。

14.2 Java 编译优化简介

Java 应用程序的编译过程与静态编译语言(例如 C 或 C++)不同。静态编译器直接把源代码转换成可以直接在目标平台上执行的机器代码，不同的硬件平台要求不同的编译器。Java 编译器把 Java 源代码转换成可移植的 JVM 字节码。我们在写代码时，常常会提到如下两条原则：

(1) 方法要尽量短，大方法要分解成小方法；

(2) 不要重复发明轮子。

我们在强调这两个原则的时候，往往只关注的是代码简洁、易维护等方便我们的因素，其实这样做还可以大大方便 Java 编译器优化代码。

Java 应用程序的编译过程与静态编译语言(例如 C 或 C++)不同。静态编译器直接把源代码转换成可以直接在目标平台上执行的机器代码，不同的硬件平台要求不同的编译器。Java 编译器把 Java 源代码转换成可移植的 JVM 字节码。与静态编译器不同，Javac 几乎不做什么优化，在静态编译语言中应当由编译器进行的优化工作，在 Java 中是在程序执行的时候，由运行时执行优化。

1. 即时编译

对于证实概念的实现来说，解释是合适的，但是早期的 JVM 由于太慢。下一代 JVM 使用即时(JIT)编译器来提高执行速度。按照严格的定义，基于 JIT 的虚拟机在执行之前，把所有字节码转换成机器码，但是以惰性方式来做这项工作：JIT 只有在确定某个代码路径将要执行的时候，才编译这个代码路径(因此有了“即时编译”的名称)。这个技术使程序能启动得更快，因为在开始执行之前，不需要冗长的编译阶段。

JIT 技术看起来很有前途，但是它有一些不足。JIT 消除了解释的负担(以额外的启动成本为代价)，但是由于若干原因，代码的优化等级仍然很一般。为了避免 Java 应用程序严重的启动延迟，JIT 编译器必须非常迅速，这意味着它无法把大量时间花在优化上。所以，早期的 JIT 编译器在进行内联假设(Inliningassumption)方面比较保守，因为它们不知道后面可能要装入哪个类。

虽然从技术上讲，基于 JIT 的虚拟机在执行字节码之前，要先编译字节码，但是 JIT 这个术语通常被用来表示任何把字节码转换成机器码的动态编译过程——即使那些能够解释字节码的过程也算。

2. HotSpot 动态编译

HotSpot 执行过程组合了编译、性能分析以及动态编译。它没有把所有要执行的字节

码转换成机器码，而是先以解释器的方式运行，只编译“热门”代码——执行得最频繁的代码。当 HotSpot 执行时，会搜集性能分析数据，用来决定哪个代码段执行得足够频繁，值得编译。只编译执行最频繁的代码有几项性能优势：没有把时间浪费在编译那些不经常执行的代码上；这样，编译器就可以花更多时间来优化热门代码路径，因为它知道在这上面花的时间物有所值。而且，通过延迟编译，编译器可以访问性能分析数据，并用这些数据来改进优化决策，例如是否需要内联某个方法调用。

为了让事情变得更复杂，HotSpot 提供了两个 Java 编译器：客户机编译器和服务器编译器。默认采用客户机编译器；在启动 JVM 时，您可以指定-server 开关，选择服务器编译器。服务器编译器针对最大峰值操作速度进行了优化，适用于需要长期运行的服务器应用程序。客户机编译器的优化目标，是减少应用程序的启动时间和内存消耗，优化的复杂程度远远低于服务器编译器，因此需要的编译时间也更少。

HotSpot 服务器编译器能够执行各种各样的类。它能够执行许多静态编译器中常见的标准优化，例如代码提升(Hoisting)、公共的子表达式清除、循环展开(Unrolling)、范围检测清除、死代码清除、数据流分析，还有各种在静态编译语言中不实用的优化技术，例如虚方法调用的聚合内联。

3．持续重新编译

HotSpot 技术另一个有趣的方面是：编译不是一个全有或者全无(All-or-Nothing)的命题。在解释代码路径一定次数之后，会把它重新编译成机器码。但是 JVM 会继续进行性能分析，而且如果认为代码路径特别热门，或者未来的性能分析数据认为存在额外的优化可能，那么还有可能用更高一级的优化重新编译代码。JVM 在一个应用程序的执行过程中，可能会把相同的字节码重新编译许多次。为了深入了解编译器做了什么，可以-XX:+PrintCompilation 标志调用 JVM，这个标志会使编译器(客户机或服务器)每次运行的时候打印一条短消息。

4．栈上(On-stack)替换

HotSpot 开始的版本编译的时候每次编译一个方法。如果某个方法的累计执行次数超过指定的循环迭代次数(在 HotSpot 的第一版中，是 10 000 次)，那么这个方法就被当作热门方法，计算的方式是：为每个方法关联一个计数器，每次执行一个后向分支时，就会递增计数器一次。但是，在方法编译之后，方法调用并没有切换到编译的版本，需要退出并重新进入方法，后续调用才会使用编译的版本。结果就是，在某些情况下，可能永远不会用到编译的版本，例如对于计算密集型程序，在这类程序中所有的计算都是在方法的一次调用中完成的。重量级方法可能被编译，但是编译的代码永远用不到。

HotSpot 最近的版本采用了称为栈上(On-stack)替换(OSR)的技术，支持在循环过程中间，从解释执行切换到编译的代码(或者从编译代码的一个版本切换到另一个版本)。

从 Java 编译、执行优化的原理可以看出，编译器会将“热门代码块”、“热门方法”持续优化，以提高性能，再回顾我们常常强调的两个原则。

(1) 尽量写小方法。小方法意味着功能单一、重用性高，自然会被很多地方用到，容易变成“热门方法”。

(2) 不重复发明轮子，尽量用已存在的轮子。大家共用一个“轮子”，自然就是“热门”轮子，Java 编译器会知道这个轮子要好好优化，让他转得更快。

14.3　Javac 编译器

Javac 编译器能够读取 Java 源代码，并将其编译成字节代码，调用 Javac 的命令行如下：

```
C:>javac options filename.java
```

此命令行中 options 选项的具体说明如下：

- -classpath path：此选项用于设定路径，在该路径上 Javac 寻找需被调用的类。该路径是一个用分号分开的目录列表。
- -d directory：此选项指定一个根目录。该目录用来创建反映软件包继承关系的目录数。
- -g：此选项在代码产生器中打开调试表，以后可凭此调试产生字节代码。
- -nowarn：此选项禁止编译器产生警告。
- -o：此选项告诉 javac 优化由内联的 static、final 以及 privite 成员函数所产生的码。
- -verbose：此选项告知 Java 显示出有关被编译的源文件和任何被调用类库的信息。

14.3.1　Javac 命令详解

Javac 有两种方法可将源代码文件名传递给 Javac：

(1) 如果源文件数量少，在命令行上列出文件名即可。

(2) 如果源文件数量多，则将源文件名列在一个文件中，名称间用空格或回车行来进行分隔。然后在 Javac 命令行中使用该列表文件名，文件名前冠以@字符。

源代码文件名称必须含有 .java 后缀，类文件名称必须含有 .class 后缀，源文件和类文件都必须有识别该类的根名。例如，名为 MyClass 的类将写在名为 MyClass.java 的源文件中，并被编译为字节码类文件 MyClass.class。

内部类定义产生附加的类文件。这些类文件的名称将内部类和外部类的名称结合在一起，例如 MyClass$MyInnerClass.class。

应当将源文件安排在反映其包树结构的目录树中。例如，如果将所有的源文件放在 /workspace 中，那么 com.mysoft.mypack.MyClass 的代码应该在 \workspace\com\mysoft\mypack.MyClass.java 中。在缺省情况下，编译器将每个类文件与其源文件放在同一目录中。可用 -d 选项(请参阅后面的选项)指定其他目标目录。工具读取用 Java 编程语言编写的类和接口定义，并将它们编译成字节码类文件。

1. 查找类型

当编译源文件时，编译器常常需要它还没有识别出的类型的有关信息。对于源文件中使用、扩展或实现的每个类或接口，编译器都需要其类型信息。这包括在源文件中没有明确提及、但通过继承提供信息的类和接口。

例如，当扩展 java.applet.Applet 时还要用到 Applet 的祖先类：java.awt.Panel、java.awt.Container、java.awt.Component 和 java.awt.Object。

当编译器需要类型信息时，它将查找定义类型的源文件或类文件。编译器先在自举类及扩展类中查找，然后在用户类路径中查找。用户类路径通过两种途径来定义：通过设置 CLASSPATH 环境变量或使用 -classpath 命令行选项(有关详细资料，请参阅设置类路径)。如果使用 -sourcepath 选项，则编译器在 sourcepath 指定的路径中查找源文件；否则，编译器将在用户类路径中查找类文件和源文件。可用-bootclasspath 和 -extdirs 选项来指定不同的自举类或扩展类，参阅下面的联编选项。

成功的类型搜索可能生成类文件、源文件或两者兼有。以下是 Javac 对各种情形进行的处理：

- 搜索结果只生成类文件而没有源文件：Javac 使用类文件。
- 搜索结果只生成源文件而没有类文件：Javac 编译源文件并使用由此生成的类文件。
- 搜索结果既生成源文件又生成类文件：确定类文件是否过时。若类文件已过时，则 Javac 重新编译源文件并使用更新后的类文件。否则，Javac 直接使用类文件。缺省情况下，只要类文件比源文件旧，javac 就认为它已过时。-Xdepend 选项指定相对来说较慢但却比较可靠的过程。

注意：Javac 可以隐式编译一些没有在命令行中提及的源文件。用 -verbose 选项可跟踪自动编译。

2. 文件列表

为缩短或简化 Javac 命令，可以指定一个或多个每行含有一个文件名的文件。在命令行中，采用@字符加上文件名的方法将它指定为文件列表。当 javac 遇到以@字符开头的参数时，它对那个文件中所含文件名的操作跟对命令行中文件名的操作是一样的。这使得 Windows 命令行长度不再受限制。

例如，可以在名为 sourcefiles 的文件中列出所有源文件的名称，该文件可能是下面的形式。

```
MyClass1.java
   MyClass2.java
   MyClass3.java
```

然后可用下列命令运行编译器：

```
C:> javac @sourcefiles
```

3. 标准选项

编译器有一批标准选项，目前的开发环境支持这些标准选项，将来的版本也将支持

它。还有一批附加的非标准选项是目前的虚拟机实现所特有的，将来可能要有变化。非标准选项以-X 打头，下面列出了 Javac 的标准选项。

(1) -classpath 类路径：设置用户类路径，它将覆盖 CLASSPATH 环境变量中的用户类路径。若既未指定 CLASSPATH 又未指定 -classpath，则用户类路径由当前目录构成。有关详细信息，请参阅设置类路径。

若未指定 -sourcepath 选项，则将在用户类路径中查找类文件和源文件。

(2) -d 目录：设置类文件的目标目录。如果某个类是一个包的组成部分，则 javac 将把该类文件放入反映包名的子目录中，必要时创建目录。例如，如果指定 -d c:\myclasses 并且该类名叫 com.mypackage.MyClass，那么类文件就叫作 c:\myclasses\com\mypackage\MyClass.class。

如果没有指定-d 选项，则 javac 将把类文件放到与源文件相同的目录中。

注意：-d 选项指定的目录不会被自动添加到用户类路径中。

(3) -deprecation：显示每种不鼓励使用的成员或类的使用或覆盖的说明。没有给出 -deprecation 选项的话，Javac 将显示这类源文件的名称：这些源文件使用或覆盖不鼓励使用的成员或类。

(4) -encoding：设置源文件编码名称，例如 EUCJIS/SJIS。若未指定 -encoding 选项，则使用平台缺省的转换器。

(5) -g：生成所有的调试信息，包括局部变量。缺省情况下，只生成行号和源文件信息。

(6) -g:none：不生成任何调试信息。

(7) -g:{关键字列表}：只生成某些类型的调试信息，这些类型由逗号分隔的关键字列表所指定。有效的关键字有：

- source：源文件调试信息。
- lines：行号调试信息。
- vars：局部变量调试信息。

(8) -nowarn：禁用警告信息。

(9) -O：优化代码以缩短执行时间。使用 -O 选项可能使编译速度下降、生成更大的类文件并使程序难以调试。在 JDK1.2 以前的版本中，Javac 的 -g 选项和 -O 选项不能一起使用。在 JDK 1.2 中，可以将-g 和-O 选项结合起来，但可能会得到意想不到的结果，如丢失变量或重新定位代码或丢失代码。-O 选项不再自动打开 -depend 或关闭 -g 选项。同样，-O 选项也不再允许进行跨类内嵌。

(10) -sourcepath 源路径：指定用以查找类或接口定义的源代码路径。与用户类路径一样，源路径项用分号“;”进行分隔，它们可以是目录、JAR 归档文件或 ZIP 归档文件。如果使用包，那么目录或归档文件中的本地路径名必须反映包名。

注意：通过类路径查找的类，如果找到了其源文件，则可能会自动被重新编译。

(11) -verbose：冗长输出。它包括了每个所加载的类和每个所编译的源文件的有关信息。

4．联编选项

在缺省情况下，类是根据与 Javac 一起发行的 JDK 自举类和扩展类来编译。但 javac 也支持联编，在联编中，类是根据其他 Java 平台实现的自举类和扩展类来进行编译的。联编时，-bootclasspath 和 -extdirs 的使用很重要。

下面列出了 Javac 的联编选项。

(1) -target 版本：生成将在指定版本的虚拟机上运行的类文件。缺省情况下生成与 1.1 和 1.2 版本的虚拟机都兼容的类文件。JDK 1.2 中的 javac 所支持的版本有：

- 1.1：保证所产生的类文件与 1.1 和 1.2 版的虚拟机兼容。这是默认状态。
- 1.2：生成的类文件可在 1.2 以后版本版的虚拟机上运行，但不能在 1.1 版的虚拟机上运行。

(2) -bootclasspath 自举类路径：根据指定的自举类集进行联编。和用户类路径一样，自举类路径项用分号“;”进行分隔，它们可以是目录、JAR 归档文件或 ZIP 归档文件。

(3) -extdirs 目录：根据指定的扩展目录进行联编。目录是以分号分隔的目录列表。在指定目录的每个 JAR 归档文件中查找类文件。

5．非标准选项

下面列出了 Javac 的非标准选项：

(1) -X：显示非标准选项的有关信息并退出。

(2) -Xdepend：递归地搜索所有可获得的类，以寻找要重编译的最新源文件。该选项将更可靠地查找需要编译的类，但会使编译进程的速度大为减慢。

(3) -Xstdout：将编译器信息送到 System.out 中。默认情况下，编译器信息送到 System.err 中。

(4) -Xverbosepath：说明如何搜索路径和标准扩展以查找源文件和类文件。

(5) -J 选项：将选项传给 javac 调用的 java 启动器。例如，-J-Xms48m 将启动内存设为 48 兆字节。虽然它不以 -X 开头，但它并不是 javac 的“标准选项”。用 -J 将选项传给执行用 Java 编写的应用程序的虚拟机是一种公共约定。

注意：CLASSPATH、-classpath、-bootclasspath 和-extdirs 并不指定用于运行 Javac 的类。如此滥用编译器的实现通常没有任何意义而且是很危险的。如果确实需要这样做，可用-J 选项将选项传给基本的 Java 启动器。

14.3.2 Javac 源码与调试

Javac 编译器不像 HotSpot 虚拟机那样使用 C++语言(包含少量 C 语言)实现，它本身就是一个由 Java 语言编写的程序，这为纯 Java 的程序员了解它的编译过程带来了很大的便利。

Javac 的源码存放在 JDK SRC HOME/langtools/src/share/classes/com/sun/tools/javac 中，除了 JDK 自身的 API 外，就只引用了 JDK SRC HOME/langtools/src/share/classes/com/sun/*里面的代码，所以调试环境建立起来简单方便，基本上不需要处理依赖关系。

以 Eclipse IDE 环境为例，先建立一个名为“Compilerj avac”的 Java 工程，然后把

JDK SRC HOME/langtools/s rc/share/classe s/com/sun/*目录下的源文件全部复制到工程的源码目录中。在导入代码期间，源码文件 AnnotationProxyMaker.java 可能会提示“Access Restriction”，被 Eclipse 拒绝编译。

这是由于 Eclipse 的 JRE System Library 中默认包含了一系列的代码访问规则(Access Rules)，如果代码中引用了这些访问规则所禁止引用的类，就会提示这个错误。可以通过添加一条允许访问 Jar 包中所有类的访问规则来解决这个问题。导入了 Javac 的源码后，就可以运行 com.sun.tools.javac.Main 的 main()方法来执行编译了，与命令行中使用 Javac 的命令没有什么区别，编译的文件与参数在 Eclipse 的 Debug Configurations 面板中的 Arguments 选项卡中指定。

虚拟机规范严格定义了 Class 文件的格式，但是对如何把 Java 源码文件转变为 Class 文件的编译过程未作任何定义，所以这部分内容是与具体 JDK 实现相关的。从 Sun Javac 的代码来看，编译过程大致可以分为三个过程，分别是：

- 解析与填充符号表过程。
- 插入式注解处理器的注解处理过程。
- 分析与字节码生成过程。

Javac 编译动作的入口是 com.sun.tools.javac.main.JavaCompiler 类，上述三个过程的代码逻辑集中在这个类的 compile()和 compile2()方法里，整个编译最关键的处理就由图中标注的 8 个方法来完成，下面我们具体看一下这 8 个方法实现了什么功能。

14.3.3 解析与填充符号表

解析步骤包括了经典程序编译原理中的词法、语法分析和填充符号表两个过程。

1) 词法、语法分析

词法分析是将源代码的字符流转变为标记(Token)集合，单个字符是程序编写过程的最小元素，而标记则是编译过程的最小元素，关键字、变量名、字面量和运算符都可以成为标记，如“int a=b+2”这句代码包含了 6 个标记，分别是 int、a、=、b、+、2，虽然关键字 int 由三个字符构成，但是它只是一个 Token，不可再拆分。在 Javac 的源码中，词法分析过程由 com.sun.tools_javac.parser.Scanner 类来实现。

语法分析是根据 Token 序列来构造抽象语法树的过程，抽象语法树(Abstract Syntax Tree，AST)是一种用来描述程序代码语法结构的树形表示方式，语法树的每一个节点都代表着程序代码中的一个语法结构(Construct)，例如包、类型、修饰符、运算符、接口、返回值甚至连代码注释等都可以是一个语法结构。

通过 Eclipse AST View 插件可以分析得出某段代码的抽象语法树视图，读者可以通过这张图对抽象语法树有一个直观的认识。在 Javac 的源码中，语法分析过程由 com.sun.tools.javac.parser.Parser 类来实现，这个阶段产出的抽象语法树由 com.sun.tools.javac.tree.JCTree 类来表示，经过这个步骤之后，编译器就基本不会再对源码文件进行操作了，后续的操作都建立在抽象语法树之上。

2) 填充符号表

完成了语法分析和词法分析之后，下一步就是填充符号表的过程，此功能是通过

enterTrees()方法实现的。符号表(Symbol Table)是由一组符号地址和符号信息构成的表格，读者可以把它想象成哈希表中 K-V 值对的形式(实际上符号表不一定是哈希表实现，可以是有序符号表、树状符号表和栈结构符号表等)。符号表中所登记的信息在编译的不同阶段都要用到。在语义分析中，符号表所登记的内容将用于语义检查(如检查一个名字的使用和原先的说明是否一致)和产生中间代码。在目标代码生成阶段，当对符号名进行地址分配时，符号表是地址分配的依据。

在 Javac 源代码中，填充符号表的过程由 com.sun.tools.j avac.comp.Enter 类实现，此过程的出口是一个待处理列表(To Do List)，包含了每一个编译单元的抽象语法树的顶级节点，以及 package-info.java(如果存在的话)的顶级节点。

14.3.4 注解处理器

JDK 1.5 之后，Java 语言提供了对注解(Annotations)的支持，这些注解与普通的 Java 代码一样，是在运行期间发挥作用的。在 JDK 1.6 中实现了 JSR-269 规范，提供了一组插入式注解处理器的标准 API，在编译期间对注解进行处理。我们可以把它看作是一组编译器的插件，在这些插件里面，可以读取、修改、添加抽象语法树中的任意元素。如果这些插件在处理注解期间对语法树进行了修改，那么编译器将回到解析及填充符号表的过程重新处理，直到所有的插入式注解处理器都没有再对语法树进行修改为止。

有了编译器注解处理的标准 API 后，我们的代码才有可能干涉编译器的行为，由于语法树中的任意元素，甚至包括代码注释都可以在插件之中访问到，所以通过插入式注解处理器实现的插件在功能上有很大的发挥空间。只要有足够的创意，程序员可以使用插入式注解处理器来实现许多原本只能在编码中完成的事情，本章最后有一个使用插入式注解处理器的简单实例。

在 Javac 源码中，插入式注解处理器的初始化过程是在 initPorcessAnnotations()方法中完成的，而它的执行过程则是在 processAnnotations()方法中完成的，这个方法判断是否还有新的注解处理器需要执行，如果有的话，则通过 com.sun.tools.javac.processing.JavacProcessingEnvironment 类的 doProcessing()方法生成一个新的 JavaCompiler 对象对编译的后续步骤进行处理。

14.3.5 语义分析与字节码生成

经过语法分析之后，编译器获得了程序代码的抽象语法树表示，语法树能表示一个结构正确的源程序的抽象，但无法保证源程序是符合逻辑的。而语义分析的主要任务是对结构上正确的源程序进行上下文有关性质的审查，如进行类型审查。举个例子，假设有如下的三个变量定义语句：

```
inta=1;
boolean b-false;
char C-2;
```

后续可能出现的赋值运算如下：

```
int d a+c;
```

```
int d=b+c;
char d=a+c;
```

后续代码中如果出现了如上三种赋值运算的话，那它们都能构成结构正确的语法树，但是只有第一种的写法在语义上是没有问题的，能够通过编译，其余两种在 Java 语言中是不合逻辑的，无法编译(是否合乎语义逻辑必须限定在具体的语言与具体的上下文环境之中才有意义。如在 C 语言中，a、b、c 的上下文定义不变，第二、三种写法都是可以被正确编译的)。

1) 检查标注

Javac 的编译过程中，语义分析过程分为标注检查和数据及控制流分析两个步骤，分别由方法 attribute()和方法 flow()完成。标注检查步骤检查的内容包括诸如变量使用前是否已被声明、变量与赋值之间的数据类型是否能够匹配，等等。在标注检查步骤中，还有一个重要的动作称为常量折叠，如果我们在代码中写了如下定义：

```
int a=1+2;
```

在语法树上仍然能看到字面量“1”、“2”和操作符“+”号，但是在经过常量折叠之后，它们将会被折叠为字面量“3”。这个插入式表达式(Infix Expression)的值已经在语法树上标注出来了(ConstantExpressionValue：3)。由于编译期间进行了常量折叠，所以在代码里面定义“a=1+2”比起直接定义“a=3”，并不会增加程序运行期哪怕仅仅一个 CPU 指令的运算量。

标注检查步骤在 Javac 源码中的实现类是 com.sun.tools.javac.comp.Attr 类和 com.sun.tools.javac.comp.Check 类。

2) 数据及控制流分析

数据及控制流分析是对程序上下文逻辑更进一步的验证，它可以检查出诸如程序局部变量在使用前是否有赋值、方法的每条路径是否都有返回值、是否所有的受查异常都被正确处理了等问题。编译时期的数据及控制流分析与类加载时的数据及控制流分析的目的基本上是一致的，但校验范围有所区别，有一些校验项只有在编译期或运行期才能进行。下面举一个关于 final 修饰符的数据及控制流分析的例子，演示代码 14-1 如下。

演示代码 14-1

```
//方法 1 有 final 修饰
public void foo(final int arg) {
    final int var = 0;
     // do something
      }
//方法 2 没有 final 修饰
public void foo(int arg) {
    final int var = 0;
   // do something
      }
```

在上述代码的两个 foo()方法中，一个方法的参数和局部变量定义使用了 final 修饰符，另外一个则没有，在代码编写时程序肯定会受到 final 修饰符的影响，不能再改变 arg 和 var 变量的值，但是这两段代码编译出来的 Class 文件是没有任何一点区别的，局部变

量与字段(实例变量、类变量)是有区别的，它在常量池中没有 CONSTANT Fieldref info 的符号引用，自然就没有访问标志(Access_Flags)的信息，甚至可能连名称都不会被保留下来(取决于编译时的选项)，自然在 Class 文件中不可能知道一个局部变量是不是被声明为 final 了。因此，将局部变量声明为 final，对运行期是没有影响的，变量的不变性仅仅由编译器在编译期间保障。在 Javac 的源码中，数据及控制流分析的入口是 flow0 方法，具体操作由 com.sun.tools.javac.comp.Flow 类来完成。

3) 解语法糖

语法糖(Syntactic Sugar)，也称糖衣语法，是由英国计算机科学家彼得•约翰•兰达(Peter J. Landin)发明的一个术语，指在计算机语言中添加的某种语法，这种语法对语言的功能并没有影响，但是更方便程序员使用。通常来说使用语法糖能够增加程序的可读性，从而减少程序代码出错的机会。

相对于 C#及许多其他 JVM 语言来说，Java 在现代编程语言之中属于“低糖语言”。尤其是 JDK 1.5 之前的版本，“低糖”语法也是 Java 语言被怀疑已经“落后”的一个表面理由。Java 中最常用的语法糖主要是前面提到过的泛型(泛型并不一定都是语法糖实现，如 C#的泛型就是直接由 CLR 支持的)、变长参数、自动装箱拆箱，等等，虚拟机运行时不支持这些语法，它们在编译阶段被还原回简单的基础语法结构，这个过程就称为解语法糖。

在 Javac 的源码中，解语法糖的过程由 desugar()方法触发，在 com.sun.tools.javac.comp.TransTypes 类和 com.sun.tools.j avac.comp.Lower 类中完成。

4) 字节码生成

字节码生成是 Javac 编译过程的最后一个阶段，在 Javac 源码里面由 com.sun.tools.javac.jvm.Gen 类来完成。字节码生成阶段不仅仅是把前面各个步骤所生成的信息(语法树、符号表)转化成字节码写到磁盘中，编译器还进行了少量的代码添加和转换工作。

例如前面章节中多次提到的实例构造器<init>()方法和类构造器<clinit>()方法就是在这个阶段被添加到语法树之中的(请注意这里的实例构造器并不是指默认构造函数，如果用户代码中没有提供任何构造函数，那编译器将会添加一个没有参数的、访问性(public、protected 或 private)与当前类一致的默认构造数，这个工作在填充符号表阶段就已经完成)，这两个构造器的产生过程实际上是一个代码收敛的过程，编译器会把语句块(对于实例构造器而言是“{}”块，对于类构造器而言是“static{}”块)、变量初始化(实例变量和类变量)、调用父类的实例构造器(仅仅是实例构造器，<clinit>()方法中无须调用父类的<clinit>()方法，虚拟机会自动保证父类构造器的执行，但在<clinit>()方法中经常会生成调用 java.lang.Obj ect 的<init>()方法的代码)等操作收敛到<init>()和<clinit>()方法之中，并且保证一定是按先执行父类的实例构造器，然后初始化变量，最后执行语句块的顺序进行，上面所述的动作由 Gen.normalizeDefs()方法来实现。除了生成构造器以外，还有其他的一些代码替换工作用于优化程序的实现逻辑，如把字符串的加操作替换为 StringBuffer 或 StringBuilder(取决于目标代码的版本是否大于或等于 JDK 1.5)的 append()操作，等等。

完成了对语法树的遍历和调整之后，就会把填充了所有所需信息的符号表交到 com.sun.tools.javac.jvm.ClassWriter 类手上，由这个类的 writeClass()方法输出字节码，生成

最终的 Class 文件，到此为止整个编译过程宣告结束。

14.3.6　Javac 编译实例

了解了 Javac 命令的基本参数的含义后，接下来将通过具体实例来演示这些参数的使用方法。

1) 编译简单程序

假设有一个源文件 Hello.java ，它定义了一个名叫 greetings.Hello 的类。greetings 目录是源文件和类文件两者的包目录，且它不是当前目录。这让我们可以使用默认的用户类路径。它也使我们没必要用 -d 选项指定单独的目标目录。

```
C:> dir
greetings/
C:> dir greetings
Hello.java
C:> cat greetings\Hello.java
package greetings;
public class Hello {
public static void main(String[] args) {
for (int i=0; i < args.length; i++) {
System.out.println("Hello " + args[i]);
}
}
}
C:> javac greetings\Hello.java
C:> dir greetings
Hello.class   Hello.java
C:> java greetings.Hello World Universe Everyone
Hello World
Hello Universe
Hello Everyone
```

2) 编译多个源文件

下面的实例可以编译 greetings 包中的所有源文件。

```
C:> dir
greetings\
C:> dir greetings
Aloha.java       GutenTag.java     Hello.java       Hi.java
C:> javac greetings\*.java
C:> dir greetings
Aloha.class       GutenTag.class     Hello.class       Hi.class
Aloha.java        GutenTag.java      Hello.java        Hi.java
```

3) 指定用户类路径

对前面实例中的某个源文件进行更改后，然后重新编译它：

```
C:> cd
\examples
C:> javac greetings\Hi.java
```

由于 greetings.Hi 引用了 greetings 包中其他的类，编译器需要找到这些其他的类。上

面的示例能运行是因为默认的用户类路径刚好是含有包目录的目录。但是，假设我们想重新编译该文件并且不关心我们在哪个目录中的话， 我们需要将“\examples”添加到用户类路径中。可以通过设置 CLASSPATH 达到此目的，但这里我们将使用 -classpath 选项来完成。

```
C:>javac -classpath \examples \examples\greetings\Hi.java
```

如果再次将 greetings.Hi 改为使用标题实用程序，该实用程序也需要通过用户类路径来进行访问：

```
C:>javac -classpath \examples:\lib\Banners.jar \
\examples\greetings\Hi.java
```

要想执行 greetings 中的类，需要访问 greetings 和它所使用的类。

```
C:>java -classpath \examples:\lib\Banners.jar greetings.Hi
```

4) 将源文件和类文件分开

将源文件和类文件置于不同的目录下经常是很有意义的，特别是在大型的项目中。我们用-d 选项来指明单独的类文件目标位置。由于源文件不在用户类路径中，所以用 -sourcepath 选项来协助编译器查找它们。

```
C:> dir
classes\  lib\     src\
C:> dir src
farewells\
C:> dir src\farewells
Base.java      GoodBye.java
C:> dir lib
Banners.jar
C:> dir classes
C:> javac -sourcepath src -classpath classes:lib\Banners.jar \
src\farewells\GoodBye.java -d classes
C:> dir classes
farewells\
C:> dir classes\farewells
Base.class     GoodBye.class
```

编译器也编译了 src\farewells\Base.java，虽然没有在命令行中指定它。要想跟踪自动编译，可使用 -verbose 选项来实现。

14.3.7 Javac 的源码与调试

在 MyEclipse 中新建一个 Java 项目，将 Javac 的源码导入进去，导入期间，源码文件 AnnotationProxyMaker.java 会报错，被 myeclipse 拒绝编译。这是由于 myeclipse 的 JRE System Library 中默认包含了一系列的代码访问规则(Access Rules)，如果代码中引用了这些访问规则所禁止引用的类，就会提示这个错误。可以通过添加一条允许访问 jar 包中所有类的访问规则来解决这个问题。

导入了 Javac 的源码之后，就可以运行 com.sun.tools.javac.Main 的 main()方法来执行编

译了，与命令行中使用 Javac 的命令没有什么区别。虚拟机规范严格定义了 Class 文件的格式，但是对如何把 Java 源码文件转变为 Class 文件的编译过程未作任何定义，所以这部分内容是与具体 JDK 实现相关的。从 Sun Javac 的代码来看，编译过程大致可以分为三个过程，分别是：

- 解析与填充符号表过程；
- 插入式注解处理器的注解处理过程；
- 分析与字节码生成过程。

Javac 编译动作的入口是 com.sun.tools.javac.main.JavaCompiler 类，上述三个过程的代码逻辑集中在这个类的 compile()和 compile2()方法里。整个编译最关键的处理是由 8 个方法来完成的，分别是：

```
initProcessAnnotations(processors);//准备过程：初始化插入式注解处理器
delegateCompiler =
    processAnootations(  //过程 2：执行注解处理
    enterTrees(stopIfError(CompileState.PARSE,    //过程 1.2：输入到符号表
    parseFiles(sourceFileObject))),      //过程 1.1：词法分析、语法分析
    classnames);

delegateCompiler.compile2();   //过程 3：分析及字节码生成
    case BY TODO:
    while(! todo.isEmpty())
    generate(desugar(flow(attribute(todo.remove()))));
    break;
    //generate,过程 3.4：生成字节码
    //desugar，过程 3.3 解语法糖
    //flow，过程 3.2：数据流分析
    //attribute，过程 3.1：标注
```

14.4　Java 语法糖的味道

几乎各种语言或多或少都提供过一些语法糖来方便程序员的代码开发，这些语法糖虽然不会提供实质性的功能改进，但是它们或能提高效率，或能提升语法的严谨性，或能减少编码出错的机会。不过也有一种观点认为语法糖并不一定都是有益的，大量添加和使用含糖的语法容易让程序员产生依赖，无法看清语法糖的糖衣背后程序代码的真实面目。总而言之，语法糖可以看做是编译器实现的一些“小把戏”，这些“小把戏”可能会使得效率有一个“大提升”，但我们也应该去了解这些“小把戏”背后的真实世界，那样才能利用好它们，而不是被它们所迷惑。

14.4.1　泛型与类型擦除

泛型是 JDK 1.5 的一项新特性，它的本质是参数化类型(Parameterized Type)的应用，也就是说所操作的数据类型被指定为一个参数。这种参数类型可以用在类、接口和方法的创建中，分别称为泛型类、泛型接口和泛型方法。

泛型思想早在 C++语言的模板(Templates)中就开始生根发芽，在 Java 语言还没有出现泛型时，只能通过 Object 是所有类型的父类和类型强制转换两个特点的配合来实现类型泛化。例如在哈希表的存取中，JDK 1.5 之前使用 HashMap 的 get()方法，返回值就是一个 Obj ect 对象，由于 Java 语言里面所有的类型都继承于 java.lang.Object，那 Object 转型成任何对象都是有可能的。但是也因为有无限的可能性，就只有程序员和运行期的虚拟机才知道这个 Object 到底是个什么类型的对象。在编译期间，编译器无法检查这个 Object 的强制转型是否成功，如果仅仅依赖程序员去保障这项操作的正确性，许多 ClassCastException 的风险就会被转嫁到程序运行期中。

泛型技术在 C#和 Java 之中的使用方式看似相同，但实现上却有着根本性的分歧，C#里面泛型无论在程序源码中、编译后的 IL 中(Intermediate Language，中间语言，这时候泛型是一个占位符)还是运行期的 CLR 中都是切实存在的，List<int>与 List<String>就是两个不同的类型，它们在系统运行期生成，有自己的虚方法表和类型数据，这种实现称为类型膨胀，基于这种方法实现的泛型被称为真实泛型。

Java 语言中的泛型则不一样，它只在程序源码中存在，在编译后的字节码文件中，就已经被替换为原来的原生类型(Raw Type，也称为裸类型)了，并且在相应的地方插入了强制转型代码，因此对于运行期的 Java 语言来说，ArrayList<int>与 ArrayList<String>就是同一个类。所以说泛型技术实际上是 Java 语言的一颗语法糖，Java 语言中的泛型实现方法称为类型擦除，基于这种方法实现的泛型被称为伪泛型。

演示代码 14-2 是一段简单的 Java 泛型例子，我们可以看一下它编译后的结果是怎样的。

演示代码 14-2

```
public static void main(String[] args) {
    Map<String, String> map = new HashMap<String, String>();
    map.put("hello", "你好");
    map.put("how are you?", "吃了没？");
    System.out.println(map.get("hello"));
    System.out.println(map.get("how are you?"));
}
```

把这段 Java 代码编译成 Class 文件，然后再用字节码反编译工具进行反编译后，将会发现泛型都不见了，程序又变回了 Java 泛型出现之前的写法，泛型类型都变回了原生类型，如演示代码 14-3 所示。

演示代码 14-3

```
public static void main(String[] args) {
    Map map = new HashMap();
    map.put("hello", "你好");
    map.put("how are you?", "吃了没？");
    System.out.println((String) map.get("hello"));
    System.out.println((String) map.get("how are you?"));
}
```

因为实现简单、兼容性考虑还是别的原因？我们已不得而知，但确实有不少人对 Java 语言提供的伪泛型颇有微词。在当时众多的批评之中，有一些是比较表面的，还有一些从

性能上说泛型会由于强制转型操作和运行期缺少针对类型的优化等原因从而导致比 C#的泛型慢一些，则是完全偏离了方向，姑且不论 Java 泛型是不是真的会比 C#泛型慢，选择从性能的角度上评价用于提升语义准确性的泛型思想但笔者也并非在为 Java 的泛型辩护，它在某些场景下确实存在不足，笔者认为通过擦除法来实现泛型丧失了泛型思想一些应有的优雅，例如演示代码 14-4 的例子。

演示代码 14-4

```
public class GenericTypes {
    public static void method(List<String> list) {
        System.out.println("invoke method(List<String> list)");
    }
    public static void method(List<Integer> list) {
        System.out.println("invoke method(List<Integer> list)");
    }
}
```

请想一想，上面这段代码是否正确，能否编译执行？也许您已经有了答案，这段代码是不能被编译的，是因为参数 List<Integer>和 List<String>编译之后都被擦除了，变成了一样的原生类型 List<E>，擦除动作导致这两个方法的特征签名变得一模一样。初步看来，无法重载的原因已经找到了，但是真的就是如此吗？只能说，泛型擦除成相同的原生类型只是无法重载的一部分原因，请再接着看一看演示代码 14-5 中的内容。

演示代码 14-5

```
public class GenericTypes {
    public static String method(List<String> list) {
        System.out.println("invoke method(List<String> list)");
        return "";
    }
    public static int method(List<Integer> list) {
        System.out.println("invoke method(List<Integer> list)");
        return 1;
    }
    public static void main(String[] args) {
        method(new ArrayList<String>());
        method(new ArrayList<Integer>());
    }
}
```

执行结果为：

```
invoke method (List<String>  list)
invoke method(List<Integer> list)
```

演示代码 14-5 与演示代码 14-4 的差别，是两个 method 方法添加了不同的返回值，由于这两个返回值的加入，方法重载居然成功了，即这段代码可以被编译和执行了。测试的时候请使用 Sun JDK 的 Javac 编译器进行编译，其他编译器，如 Eclipse JDT 的 ECJ 编译器，仍然可能会拒绝编译这段代码，ECJ 编译时会提示“Method method(List<String>has the same erasure method(List<E>as another method in type GenericTypes”。

这是我们对 Java 语言中返回值不参与重载选择的基本认知的挑战吗？演示代码 14-5

中的重载当然不是根据返回值来确定的，之所以这次能编译和执行成功，是因为两个mehtod()方法加入了不同的返回值后才能共存在一个Class文件之中。因为方法重载要求方法具备不同的特征签名，返回值并不包含在方法的特征签名之中，所以返回值不参与重载选择，但是在Class文件格式之中，只要描述符不是完全一致的两个方法就可以共存。也就是说两个方法如果有相同的名称和特征签名，但返回值不同，那它们也是可以合法地共存于一个Class文件中的。

由于Java泛型的引入，各种场景(虚拟机解析、反射等)下的方法调用都可能对原有的基础产生影响和新的需求，如在泛型类中如何获取传人的参数化类型等。所以JCP组织对虚拟机规范做出了相应的修改，引入了诸如Signature和LocalVariableTypeTable等新的属性用于解决伴随泛型而来的参数类型的识别问题，Signature是其中最重要的一项属性，它的作用就是存储一个方法在字节码层面的特征签名，这个属性中保存的参数类型并不是原生类型，而是包括了参数化类型的信息。修改后的虚拟机规范要求所有能识别49.0以上版本的Class文件的虚拟机都要能正确地识别Signature参数。

从上面的例子可以看到擦除法对实际编码带来的影响，由于List<String>和List<Integer>擦除后是同一个类型，我们只能添加两个并不需要实际使用到的返回值才能完成重载，这是一种毫无优雅和美感可言的解决方案。同时，从Signature属性的出现我们还可以得出结论，擦除法所谓的擦除，仅仅是对方法的Code属性中的字节码进行擦除，实际上元数据中还是保留了泛型信息，这也是我们能通过反射手段取得参数化类型的根本依据。

14.4.2 自动装箱、拆箱与遍历循环

就纯技术的角度而论，自动装箱、自动拆箱与遍历循环(Foreach循环)这些语法糖，无论是实现上还是思想上都不能和上一节介绍的泛型相比，两者的难度和深度都有很大的差距。专门拿出一节来讲解它们只有一个理由：毫无疑问，它们是Java语言里面被使用得最多的语法糖。我们通过下面的演示代码14-6和演示代码14-7，来看看这些语法糖在编译后会发生什么样的变化。

演示代码 14-6

```
public static void main(String[] args) {
   List<Integer> list = Arrays.asList(1, 2, 3, 4);
         //  rtpxA JDK 1.7 中还有另外一颗语法糖
      //可以让上述代码进一步简化为List<Integer> list = [1,2,3,4];
    int SUTI1 = 0;
  for (int i : list) {
      sum += i;
           }
          Systam.out.println (sum) ;
      }
```

演示代码 14-7

```
public static void main(String[l args) {
    List list ~ Arrays.asList( new Integerf] {
          Integer.valueOf (1) ,
```

```
                    Integer.valueOf ( 2 ) ,
             Integer.valueOf (3) ,
             Integer.valueOf (4)  }) ;
     int SUIT1 = 0;
    for (Iterator locaiiterator = list.iterator();
localIterator.hasNext(); ) {
          int  i  =  ( (Integer)localIterator.next () ) .intValue () ;
      sum += i,
             }
         System.out.println (sum) ;
    }
```

演示代码 14-6 中共包含了泛型、自动装箱、自动拆箱、遍历循环与变长参数 5 种语法糖，演示代码 14-7 则展示了它们在编译后的变化。泛型就不必说了，自动装箱、拆箱在编译之后被转化成了对应的包装和还原方法，如本例中的 Integer.valueOf()与 Integer.intValue()方法，而遍历循环则是把代码还原成了迭代器的实现，这也是为何遍历循环需要被遍历的类实现 Iterable 接口的原因。最后再看看变长参数，它在调用的时候变成了一个数组类型的参数，在变长参数出现之前，程序员就是使用数组来完成类似功能的。

这些语法糖虽然看起来很简单，但也不见得就没有任何值得我们注意的地方，演示代码 14-8 演示了自动装箱的一些错误用法。

演示代码 14-8

```
  public static void main(String[] args) {
     Integer a = 1;
     Integer b = 2;
     Integer c = 3;
     Integer d = 3;
     Integer e = 321;
     Integer f = 321;
    Long g = 3L;
          System.out.println (c  = =  d) ;
       System.out.println(e -N f) ,
        System.out.println(c  = =  (a + b) ) ;
         System.out.println (c.equals (a  +  b) ) ;
        System.out.println(g ==  (a + b) ) ;
         System.out.println (g.equals (a  +  b) ) ;
     }
```

看完演示代码 14-8，不妨思考两个问题：一是代码中的这 6 句打印语句的输出是什么？二是这六句打印语句中，解除语法糖后参数会是什么样子？这两个问题的答案很容易就都能试验出来，笔者在此暂且略去答案，希望读者自己上机实践一下。无论你的回答是否正确，鉴于包装类的“==”运算在没有遇到算术运算的情况下不会自动拆箱，而且它们的 equals()方法不会处理数据转型的关系，笔者建议在实际编码中应尽量避免这样来使用自动装箱与拆箱。

14.4.3　条件编译

许多程序设计语言都提供了条件编译的途径，如 C、C++中使用预处理器指示符

(#ifdef)来完成条件编译。C、C++的预处理器最初的任务是解决编译时的代码依赖关系(众所周知的#include 预处理命令)，而在 Java 语言之中并没有使用预处理器，因为 Java 语言天然的编译方式(编译器并非一个一个地编译 Java 文件，而是将所有的编译单元的语法树顶级节点输入到待处理列表后再进行编译，因此各个文件之间能够互相提供符号信息)无须使用预处理器。那 Java 语言是否有办法实现条件编译呢？

Java 语言当然也可以进行条件编译，方法就是使用条件为常量的 if 语句。如演示代码 14-9 所示，此代码中的 if 语句不同于其他 Java 代码，它在编译阶段就会被“运行”，生成的字节码之中只包括“System.out.println(” block 1”)； ” 一条语句，并不会包含 if 语句及另外一个分子中的“System.out.println(“block 2”);” 。

演示代码 14-9

```
public static void main(String[] args) {
  if (true) {
              System.out.println ( "block  1ii) ;
  } else {
                System.out.println ( "block  2 " ) ;
            }
        }
```

此代码编译后 Class 文件的反编译结果：

```
public static void main(String[] args)  {
System.out.println( "block  l'i);
}
```

只能使用条件为常量的 if 语句才能达到上述效果，如果使用常量与其他带有条件判断能力的语句搭配，则可能在控制流分析中提示错误，被拒绝编译，如演示代码 14-10 所示的代码就会被编译器拒绝编译，演示代码 14-10 不能使用其他条件语句来完成条件编译。

演示代码 14-10

```
public static void main (String[] args)  {
//编译器将会提示"Unreachable code"
while (false)  {
       System.out.println('''') ;
    }
```

Java 语言中条件编译的实现，也是 Java 语言的一颗语法糖，根据布尔常量值的真假，编译器将会把分支中不成立的代码块消除掉，这一工作将在编译器解除语法糖的阶段(com.sun.tools.javac.comp.Lower 类中)完成。由于这种条件编译的实现方式使用了 if 语句，所以它必须遵循最基本的 Java 语法，只能写在方法体内部，因此它只能实现语句基本块(Block)级别的条件编译，而没有办法实现根据条件调整整个 Java 类的结构。

除了本节中介绍的泛型、自动装箱、自动拆箱、遍历循环、变长参数和条件编译之外，Java 语言还有不少其他的语法糖，如内部类、枚举类、断言语句、对枚举和字符串(在 JDK 1.7 中支持)的 switch 支持、在 try 语句中定义和关闭资源(在 JDK 1.7 中支持)等，你可以通过跟踪 Javac 源码、反编译 Class 文件等方式了解他们的本质实现。

14.5　插入式注解处理器

插入式注解处理 API(JSR 269)提供一套标准 API 来处理 Annotations(JSR 175)，是从 JDK 1.6 开始推出的一种机制。在本节将简单介绍插入式注解处理 API 的基本知识，并实战演练了使用插入式注解处理 API 的基本过程。

14.5.1　插入式注解处理 API 基础

实际上 JSR 269 不仅仅用来处理 Annotation，其更强大的功能是它建立了 Java 语言本身的一个模型，它把 method、package、constructor、type、variable、enum、annotation 等 Java 语言元素映射为 Types 和 Elements(两者有什么区别?)，从而将 Java 语言的语义映射成为对象，我们可以在 javax.lang.model 包下面可以看到这些类。所以我们可以利用 JSR 269 提供的 API 来构建一个功能丰富的元编程(Metaprogramming)环境。JSR 269 用 Annotation Processor 在编译期间而不是运行期间处理 Annotation，Annotation Processor 相当于编译器的一个插件，所以称为插入式注解处理。如果 Annotation Processor 处理 Annotation 时(执行 process 方法)产生了新的 Java 代码，编译器会再调用一次 Annotation Processor，如果第二次处理还有新代码产生，就会接着调用 Annotation Processor，直到没有新代码产生为止。每执行一次 process()方法被称为一个“round”，这样整个 Annotation processing 过程可以看作是一个 round 的序列。JSR 269 主要被设计成为针对 Tools 或者容器的 API。举个例子，我们想建立一套基于 Annotation 的单元测试框架(如 TestNG)，在测试类里面用 Annotation 来标识测试期间需要执行的测试方法如下：

```
@TestMethod
 public void testCheckName(){
      //do something here
 }
```

这时我们就可以用 JSR 269 提供的 API 来处理测试类，根据 Annotation 提取出需要执行的测试方法。

另一个例子是如果我们出于某种原因需要自行开发一个符合 Java EE 5.0 的 Application Server(当然不建议这样做)，我们就必须处理 Common Annotations(JSR 250)、Web Services Metadata(JSR 181)等规范的 Annotations，这时可以用 JSR 269 提供的 API 来处理这些 Annotations。在现在的开发工具里面，Eclipse 支持 JSR 269。

下面演示如何来用 JSR 269 提供的 API 来处理 Annotations 和读取 Java 源文件的元数据(Metadata)。

```
@SupportedAnnotationTypes("PluggableAPT.ToBeTested")//可以用"*"表示支持所有
Annotations
@SupportedSourceVersion(SourceVersion.RELEASE_6)
public class MyAnnotationProcessor extends AbstractProcessor {
    private void note(String msg) {
        processingEnv.getMessager().printMessage(Diagnostic.Kind.NOTE, msg);
    }
```

```
    public boolean process(Set<? extends TypeElement> annotations,
RoundEnvironment roundEnv) {
        //annotations 的值是通过@SupportedAnnotationTypes 声明的且目标源代码拥有
的所有 Annotations
        for(TypeElement te:annotations){
            note("annotation:"+te.toString());
        }
        Set<? extends Element> elements = roundEnv.getRootElements();//获
取源代码的映射对象
          for(Element e:elements){
            //获取源代码对象的成员
            List<? extends Element> enclosedElems =
e.getEnclosedElements();
            //留下方法成员,过滤掉其他成员
            List<? extends ExecutableElement> ees =
ElementFilter.methodsIn(enclosedElems);
            for(ExecutableElement ee:ees){
                note("--ExecutableElement name is "+ee.getSimpleName());
                List<? extends AnnotationMirror> as =
ee.getAnnotationMirrors();//获取方法的 Annotations
                note("--as="+as);
                for(AnnotationMirror am:as){
                    //获取 Annotation 的值
                    Map<? extends ExecutableElement, ? extends
AnnotationValue> map= am.getElementValues();
                    Set<? extends ExecutableElement> ks = map.keySet();
                    for(ExecutableElement k:ks){//打印 Annotation 的每个值
                        AnnotationValue av = map.get(k);
                        note("----
"+ee.getSimpleName()+"."+k.getSimpleName()+"="+av.getValue());
                    }
                }
            }
        }
        return false;
    }
}
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface ToBeTested{
    String owner() default "Chinajash";
    String group();
}
```

编译以上代码，然后再创建下面的 Testing 对象，不要编译 Testing 对象。

```
public class Testing{
    @ToBeTested(group="A")
    public void m1(){
    }
    @ToBeTested(group="B",owner="QQ")
    public void m2(){
    }
    @PostConstruct//Common Annotation 里面的一个 Annotation
    public void m3(){
```

```
    }
}
```

下面用以下命令来编译 Testing 对象：

```
javac -XprintRounds -processor PluggableAPT.MyAnnotationProcessor Testing.java
```

-XprintRounds 表示打印 round 的次数，运行上面命令后在控制台会看到如下输出：

```
Round 1:
        input files: {PluggableAPT.Testing}
        annotations: [PluggableAPT.ToBeTested,
javax.annotation.PostConstruct]
        last round: false
Note: annotation:PluggableAPT.ToBeTested
Note: --ExecutableElement name is m1
Note: --as=@PluggableAPT.ToBeTested(group="A")
Note: ----m1.group=A
Note: --ExecutableElement name is m2
Note: --as=@PluggableAPT.ToBeTested(group="B", owner="QQ")
Note: ----m2.group=B
Note: ----m2.owner=QQ
Note: --ExecutableElement name is m3
Note: --as=@javax.annotation.PostConstruct
Round 2:
        input files: {}
        annotations: []
        last round: true
```

通过阅读 Javac 编译器的源码，我们知道编译器在把 Java 程序源码编译为字节码的时候，会对 Java 程序源码做各方面的检查校验。这些校验主要以程序“写得对不对”为出发点，虽然也有各种 WARNING 的信息，但总体来讲还是较少去校验程序“写得好不好”。有鉴于此，业界出现了许多针对程序“写得好不好”的辅助校验工具，如 CheckStyle、FindBug、Klocwork 等。这些代码校验工具有一些是基于 Java 的源码进行校验，有一些是通过扫描字节码来完成，在本节的实战中，我们将会使用注解处理器 API 来编写一款拥有自己编码风格的校验工具 NameCheckProcessor。

当然，由于我们的实战都是为了学习和演示技术原理，而不是为了做出一款能媲美 CheckStyle 等工具的产品来，所以 NameCheckProcessor 的目标也仅定为对 Java 程序命名进行检查，根据《Java 语言规范》中的要求，Java 程序命名应当符合下列格式的书写规范：

- 类(或接口)：符合驼式命名法，首字母大写。
- 方法：符合驼式命名法，首字母小写。
- 字段：符合驼式命名法，首字母小写。
- 类或实例变量：符合驼式命名法，首字母小写。
- 常量：要求全部由大写字母或下划线构成，并且第一个字符不能是下划线。

上文提到的驼式命名法(Camel Case Names)，正如它的名称所表示的那样，是指混合使用大小写字母来分割构成变量或函数的名字，犹如驼峰一般，这是当前 Java 语言中主流的命名规范，我们的实战目标就是为 Javac 编译器添加一个额外的功能，在编译程序时检

查程序名是否符合上述对类(或接口)、方法、字段的命名要求。

14.5.2 实战

要通过注解处理 API 实现一个编译器插件，首先需要了解这组 API 的一些基本知识。我们实现注解处理器的代码需要继承抽象类 javax.annotation.processing.AbstractProcessor，这个抽象类中只有一个必须覆盖的 abstract 方法“process()”，它是 Javac 编译器在执行注解处理器代码时要调用的过程，我们可以从这个方法的第一个参数“annotations”中获取到此注解处理器所要处理的注解集合，从第二个参数“roundEnv”中访问到当前这个 Round 中的语法树节点，每个语法树节点在这里表示为一个 Element。在 JDK 1.6 新增的 javax.lang.model 包中定义了 16 类 Element，包括了 Java 代码中最常用的元素，例如“包(PACKAGE)、枚举(ENUM)、类(CLASS)、注解(ANNOTATION_TYPE)、接口(INTERFACE)、枚举值(ENUM_CONSTANT)、字段(FIELD)、参数(PARAMETER)、本地变量(LOCAL_VARIABLE)、异常 (EXCEPTION_PARAMETER)、方法(METHOD)、构造函数(CONSTRUCTOR)、静态语句块(STATIC INIT，即 static{}块)、实例语句块(INSTANCE—INIT，即{}块)、参数化类型(TYPE—PARAMETER，即泛型尖括号内的类型)和未定义的其他语法树节点(OTHER)”。除了 process()方法的传入参数之外，还有一个很常用的实例变量“processingEnv”，它是 AbstractProcessor 中的一个 protected 变量，在注解处理器初始化的时候(init()方法执行的时候)创建，继承了 AbstractProcessor 的注解处理器代码可以直接访问到它。它代表了注解处理器框架提供的一个上下文环境，要创建新的代码、向编译器输出信息、获取其他工具类等都需要用到这个实例变量。

注解处理器除了 process()方法及其参数之外，还有两个可以配合使用的 Annotations：@SupportedAnnotationTypes 和@SupportedSourceVersion，前者代表了这个注解处理器对哪些注解感兴趣，可以使用星号“木”作为通配符代表对所有的注解都感兴趣，后者指出这个注解处理器可以处理哪些版本的 Java 代码。

每一个注解处理器在运行的时候都是单例的，如果不需要改变或生成语法树的内容，process()方法就可以返回一个值为 false 的布尔值，通知编译器这个 Round 中的代码未发生变化，无需构造新的 JavaCompiler 实例，在这次实战的注解处理器中只对程序命名进行检查，不需要改变语法树的内容，因此 process()方法的返回值都是 false。关于注解处理器的 API，笔者就简单介绍这些，对这个领域有兴趣的读者可以阅读相关的帮助文档。我们来看看注解处理器 NameCheckProcessor 的具体代码，如演示代码 14-11 所示。

演示代码 14-11

```
@SupportedAnnotationType t3 { " * " )
@SupportedSourceVersion ( SourcaVersion . RELEASE 6 )
public class NameCheckProcessor extends AbstractProcessor {

  private NameChecker nameChecker;
@Override
  public void init(ProcessingEnvironment processingEnv) {
        super.init (processingEnv) ;
       nameChecker = new NameChecker{processingEnv) ;
```

```
        }
          @Over ride
    public boolean process(Set<'l extends TypeElement> annotationa,
    RoundEnvironment roundEnv) {
            if  (lroundEnv.processingOrrer ()) {
           for (Element element : roundEnv.getRootElements())
                                          nameChecker. checkNames
( alement)  ;
                     }
        return false,
         }
}
```

从上面的代码可以看到 NameCheckProcessor 能处理基于 JDK 1.6 的源码，它不限于特定的注解，对任何代码都“感兴趣”，而在 process()方法中是把当前 Round 中的每一个 RootElement 传递到一个名为 NameChecker 的检查器中执行名称检查逻辑，NameChecker 的代码如演示代码 14-12。

演示代码 14-12

```
 public class NameChecker {
      private final Messager messager;
         NameCheckScanner nameCheckScanner = new NameCheckScanner() ;
      NameChecker(ProcessingEnvironment processsingEnv) {
               this.messager = processsingEnv.getMessager() ;
                   }
      public void checkNames(Element element) {
                             nameCheckScanner. scan ( element)  ;
                  }
     private class NameCheckScanner extends ElementScanner6<Void, Void>
{
@Override
         public Void visitType(TypeElement e, Void p) {
                             scan (e.getTypeParameters ( ) ,   p) ;
                       checkCamelCase (e,  true) ;
                    super.visitType (e,  p) ;
              return null;
                    }
 @Override
          public Void visitExecutable(ExecutableElement e, Void p) {
            if (e.getKind() == METHOD) {
                  Name name = e.getSimpleName () ;
                  if   (name. contentEquals (e .getEnclosingElement
( ) .getSimpleName ( ) ) )
                 messager.printMessage(WARNING, " -   iKik- "" + name +
11" TBl
                                       ~ k Jil 91~,  i~t k#git.&4tcn.12iir
",  e) ;
                              checkCamelCase (e,  false) ;
                                 }
                        super.visitExecutable (e,  p) ;
               return null;
                              }
@Override
```

```
          public Void visitVariable(VariableElement e, Void p) {
            if (e.getKind() == ENUM CONSTANT lI e.getConstantValue() l-
null lI
                   heuristicallyConstant (e) )
                               checkAllCaps (e) ;
    else
                                  checkCamelCase (e,  false) ;
               return null;
                         }
         private boolean heuristicallyConstant(VariableElement e) {
                    if  (e.getEnclosingElement () .getKind()  == INTERFACE).
                    return true;
                    else if  (e.getKind()  == FIELD &&
e.getModifiers() .containsAll(EnumSet.
of (PUBLIC,  STATIC,  FINAL) ) )
                    return true,
           else {
                   return false;
                         }
                    }
private void checkCamelCase(Element e. boolean initialCaps) {
       String name  =  e.getSimpleName () .toString () ;
   bodlean previousUpper = false;
   boolean conventional = true;
      int  firstCodePoint  = name.codePointAt (O) ,
      if  (Character.isUpperCase (firstCodePoint) )  {
      previousUpper = true;
     if {!initialCaps) {
       messager.printMessage (WARNING, " .g*tS " " +  name + " " ,8t
b },X,J.~ll
             ig:lt+ ", e) ;
  return;
                  }
    } else if (Character.isLowerCase(firstCodePoint)) {
     if (initialCaps) {
       messager.printMessage (WARNING.  " .9-f*: " "  + name  +  " " &
WA+\:n
                    *jf-* ",  e) ;    . .
  return,
                  }
   } else
      conventional = false;
   if (conventional) {
      int cp = firstCodePoint;
          for  (int i = Character.charCount (cp) ;  i < name.length();  i
+z
           Character.charCount (cp) )  {
                cp = name.codePointAt (i) ;
                  if  (Character.isUpperCase (cp) )  {
           if (previousUpper) {
                            . conventional  =  false;
                                                  break ;
                             }
```

```
            previousUpper — true;
        } else
            previousUpper = false;
                }
        }
        if   ( ! conventional)
            messager.printMessage (WARNING,  " )9+f. "" + name + "" La!
~t~~e-1.4~.91L-
            (Camel Case Names) ",  e) ;
                }
            private void checkAllCaps (Element e)  {
                    String name  =  e.getSimpleNarTie () .toString () ;
            boolean conventional = true;
                int firstCodePoint = name.codePointAt (O) ;
                        if   ( ! Character.isUpperCase
( firstCodePoint) )
            conventional = false;
        else {
            boolean previousUnderscore = false;
            int cp = firstCodePoint;
              for  (int i = Character.charCount(cp) ;  i < name.length();
i += .
                    Character.charCount (cp) )  {
                        cp = name.codePointAt (i) ;
            if (cp == (int) ' ') {
                if (previousUnderscore) {
                    conventional = false;
    break;
                                        }
                previousUnderscore = true;
            } else {
                previousUnderscore = false;
                if  ( !Character.isUpperCase (cp)  && !Character.isDigit
(cp) ){
                    conventional = false;
    break;
                            }
                        }
                    }
                }
                if   ( ! conventional)
                    messager.printMessage (WARNING, "常量"  +  name  +
" "应当全部以大写字母或下划线命名，并且以字母开头",e) ;
                }
        }
}
```

NameChecker 的代码看起来有点长，但实际上注释占了很大 -部分，而且即使算上注释也不到 190 行。它通过一个继承于 javax.lang.mo del.util.ElementS canner6 的 NameCheck Scanner 类，以 Visitor 模式来完成对语法树的遍历，分别执行 visitType()、visitVariable()和 visitExecutable()方法来访问类、字段和方法，这 3 个 visit 方法对各自的命名规则做相应的检查，checkCamelCase()与 checkAllCaps()方法则用于实现驼式命名法和全大写命名规则的检查。

整个注解处理器只需 NameCheckProcessor 和 NameChecker 两个类就可以全部完成，为了验证我们的实战成果，在下面的代码中提供了一段命名规范的“反面教材”代码，其中的每一个类、方法及字段的命名都存在问题，但是使用普通的 Javac 编译这段代码时不会提示任意一个 Waming 信息。

```
  public class BADLY NAMED CODE {
     enum colors {
        red, blue, green;
              }
     static final int  FORTY TWO = 42;
     public static int NOT A CONSTANT =  FORTY TWO;
      protected void BADLY NAMED CODE() {
 return;
             }
     public void NOTcamelCASEmethodNAME() {
 return;
             }
       }
```

接下来开始运行并测试前面的程序。我们可以通过 Javac 命令的“-processor”参数来执行编译时需要附带的注解处理器，如果有多个注解处理器的话，用逗号分隔。还可以使用-XprintRounds 和-XprintProcessorInfo 参数来查看注解处理器运作的详细信息，本次实战中的 NameCheckProcessor 的编译及执行过程如下。

```
           D: \ src >j avac  org/fenixsoft/compile/NameChecker . j ava
           D : \src >j avac  org/fenixsoft/compile/NameCheckProcessor . j ava
  D:\src>javac -processor org.fenixsoft.compile.NameCheckProcessor
org/fenixsoft/
 compile/BADLY NAMED CODE . java
           org\ fenixsoft\compile\BADLY NAMED CODE . java: 3 :    Y-4-  : k #s
n BADLY NAMED CODE " EL % .ff g
gt1:A4~.9ib (Camel Case Names)
   public class BADLY NAMED CODE {
             A
           org\ fenixsoft \compile\BADLY NAMED CODE . java: 5 :   %%  : k+i:
" colors'~ ~ vX k:~ ~kt 9~
        enum colors {
              A
           org\ fenixsoft \compile\BADLY  NAMED CODE . java : 6 :   %4- ~jt  "
red"        b4F vX k:~ %-&-iTXi]
 t~4y-g ,  +lLvX q-{kif'91
                 red, blue, green,
                       A
    org\fenixsoft\compile\BADLY NAMED CODE.java:6:  y~-: $'f "blue"
FvXk:nk~T
3V]lX4it-g, +1I~vXq-itjt9,
                 red, blue, green;
                           A
            org\ fenixsoft\compile\BADLY NAMED CODE . java: 6 :   ~-         -
"green" ,S  ~ k4F vA k ~  l it 4',:T
3i}l~4~-g, if- EL},X~~g:it9~
                 red, blue, green;
                                 A
```

```
          org\ fenixsoft \compile\BADLY NAMED CODE . java : 9 :    'V%   u
FORTY TWO " lt q* ~4F r A k k'q
-8: alT3cij1#e4~-9 ,  if- a.vX 719:if-9;
        static final int  FORTY TWO = 42;
                                A
           org\ f enixsoft \ compile \BADLY NAMED CODE . java :11:    sl:4-
: h -fl:  u NOT A CONSTANT " )tjL % vA ,J-:~
 %~:jf- 91
        public static int NOT A CONSTANT =  FORTY TWO;
                               A
          org\ fenixsoft \compile\BADLY NAMED CODE. java :13 :   f % : h+i:
"Test " BQi vx,J.:nk-lt 3k
            protected void Test () {
                               A
     org\fenixsoft\compile\BADLY NAMED CODE.java:17:  +4- : h k
"NOTcamelCASEmethodNAME"
  EL b vX,J- :~ q~:jr- ik

  public void NOTcamelCASEmethodNAME() f
```

NameCheckProcessor 的实战例子只演示了 JSR-269 嵌入式注解处理 API 其中的一部分功能，基于这组 API 支持的项目还有用于校验 Hibernate 标签使用正确性的 Hibernate Validator Annotation Processor(本质上与 NameCheckProcessor 所做的事情差不多)、自动为字段生成 getter 和 setter 方法的 Project Lombok(根据已有元素生成新的语法树元素)等，读者有兴趣的话可以参考它们官方站点的相关内容。

第 15 章



运行期优化

上一章讲解了编译优化的基本知识。本章将进一步讲解 Java 技术的优化知识，详细讲解 JVM 在程序运行时期的优化技术方案，为读者学习后面的知识打下基础。

15.1 运行期优化简介

Java 编程语言是一种新的具有独特性能特征的编程语言。迄今为止，大部分试图提高其性能的尝试都集中在如何应用为传统语言开发的编译技术上。及时编译器是基本的 快速传统编译器，它可以"在运行中"将 Java 字节码转换为本地机器代码。及时编译器在终端用户的实际执行字节码的机器上运行，并编译每一个被首次执行的方法。

在部分的商用虚拟机(Sun HotSpot、IBM J9)中，Java 程序最初是通过解释器(Interpreter)进行解释执行的，当虚拟机发现某个方法或代码块的运行特别频繁，就会把这些代码认定为"热点代码"(Hot Spot Code)，为了提高热点代码的执行效率，在运行时，虚拟机将会把这些代码编译成与本地平台相关的机器码，并进行各种层次的优化，完成这个任务的编译器称为即时编译器(Just In Time Compiler，下文中简称 JIT 编译器)。

即时编译器并不是虚拟机必需的部分，Java 虚拟机规范并没有规定 Java 虚拟机内必须要有即时编译器，更没有限定或指导即时编译器应该如何去实现。但是，即时编译器编译性能的好坏、代码优化程度的高低却是衡量一款商用虚拟机优秀与否的最关键的指标之一，它也是虚拟机中最核心最能体现技术水平的部分。在本章中，我们将走进虚拟机的内部，探索即时编译器的运作过程。

由于 Java 虚拟机规范没有具体的约束规则去限制即时编译器应该如何实现，所以这部分功能完全是与虚拟机具体实现(Implementation Specific)相关的内容，如无特殊说明，本章中所提及的编译器、即时编译器都是指 HotSpot 虚拟机内的即时编译器，虚拟机也是特指 HotSpot 虚拟机。不过，本章中的大部分内容是描述即时编译器的行为，涉及编译器实现层面的内容较少，而主流虚拟机中即时编译器的行为又有很多相似相通之处，因此对其他虚拟机来说也具备较大的参考意义。

15.2 HotSpot 虚拟机内的即时编译器

在本节的内容中，我们将要了解 HotSpot 虚拟机内的即时编译器的运作过程，同时，我们要解决以下几个问题：

- 为何 HotSpot 虚拟机要使用解释器与编译器并存的架构？
- 为何 HotSpot 虚拟机要实现两个不同的即时编译器？
- 程序何时使用解释器执行？何时使用编译器执行？
- 哪些程序代码会被编译为本地代码？如何编译本地代码？
- 如何从外部观察即时编译器的编译过程和编译结果？

15.2.1 HotSpot 虚拟机的背景

在 JIT 编译中存在着几个问题。首先，由于编译器是在"用户时间"内运行于执行字

节码的机器上，因此它将受到编译速度的严格限制：如果编译速度不是特别快，则用户将会感到在程序的启动或某一部分的明显的延迟。这就不得不采取一种折中方案，用这种折中方案将很难进行最好的优化，从而将会大大地降低编译性能。

其次，即使 JIT 有时间进行全优化，这样的优化对 Java 编程语言来说，也比对传统语言(如 C 和 C++)的优化效果要差。这有以下几个原因。

(1) Java 编程语言是动态“安全的”，其含义是保证程序不违反语言的语义或直接访问非结构化内存。这就意味着必须经常进行动态类型测试，例如，当转型时(Casting)和向对象数组进行存储时。

(2) Java 编程语言在“堆(heap)”上对所有对象进行分配，而在 C++中，许多对象是在栈(Stack)上分配的。这就意味着 Java 编程语言的对象分配效率比 C++的对象分配效率要高得多。除此之外，由于 Java 编程语言是垃圾回收式的，因而它比 C++有更多的不同类型的内存分配开销(包括潜在的垃圾清理(Scavenging)和编写-隔离(Write-barrier)开销)。

(3) 在 Java 编程语言中，大部分方法调用是“虚拟的”(潜在是多态的)，这在 C++中很少见。这不仅意味着方法调用的性能更重要，而且意味着更难以为方法调用而执行静态编译器优化(特别是像内嵌方法(Inlining)那样的全局优化)。大多数传统优化在调用之间是最有效的，而 Java 编程语言中的减小的调用间距离可大大降低这种优化的效率，这是因为它们使用了较小的代码段的缘故。

(4) 基于 Java 技术的程序由于其强大的动态类装载的能力，因而可“在运行中”发生改变。这就使得它特别难于进行许多类型的全局优化，因为编译器不仅必须能够检测这些优化何时会由于动态装载而无效，而且还必须能够在程序执行过程中解除和/或重做这些优化，且不会以任何形式损坏或影响基于 Java 技术的程序的执行语义(即使这些优化涉及栈上的活动方法)。

上述问题的结果是使得任何试图获取 Java 编程语言的先进性能的尝试，都必须寻求一种非传统的解决方案，而不是盲目地应用传统编译器技术。Java HotSpot 性能引擎的体系结构通过使用适配性的优化技术，解决了以上所提出的 Java 编程语言的性能问题。适配性的优化技术是 Sun 公司的研究机构 Self 小组多年以来在面向对象的语言实现上的研究成果。

1. 热点 Hot Spot 检测

适配性的优化技术利用了大多数程序的有趣的属性，解决了 JIT 编译问题。实际上，所有程序都是花费了它们的大部分时间而执行了它们中的很小一部分代码。Java HotSpot 性能引擎不是在程序一启动时就对整个程序进行编译，而是在程序一启动时就立即使用解释器(Interpreter)运行该程序，在运行中对该程序进行分析以检测程序中的关键性“热点(Hot Spot)”，然后，再将全局本地码(Native-code)优化器集中在这些热点上。通过避免编译(大部分程序的)不常执行的代码，Java HotSpot 编译器将更多的注意集中于程序的性能关键性部分，因而不必增加总的编译时间。这种动态监测随着程序的运行而不断进行，因而，它可以精确地“在运行中”调整它的性能以适应用户的需要。

这种方法的一个巧妙而重要的益处是，通过将编译延迟到代码已被执行一会儿之后，从而可在代码被使用的过程中收集信息，并使用这些信息进行更智能的优化。除收集程序

中的热点信息外，也收集其他类型的信息，如与“虚拟”方法调用有关的调用者 被调用者的关系数据等。

2．方法内嵌

正像在“背景说明”中所提到的，Java 编程语言中的“虚拟”方法调用的出现频率，是一个重要的妨碍优化的瓶颈。当 Java HotSpot 适配性优化器在执行过程中，一旦回收了有关程序“热点”的信息后，它不仅能将这些“热点”编译为本地代码，而且还可以执行内嵌在这些代码上的大量的方法。

内嵌具有重要的益处。它极大地减小了方法调用的动态频率，这就节省了执行这些方法调用所需要的时间。而更重要的是，内嵌为优化器生成了大得多的代码块。这种状态可以大大地提高传统编译器的优化技术的效率，从而消除提高 Java 编程语言性能的障碍。

内嵌对其他代码的优化起到了促进作用，它使优化的效率大大提高。随着 Java HotSpot 编译器的进一步成熟，操作更大的内嵌代码块的能力将使实现更先进的优化技术成为可能。

3．动态逆优化

尽管上述内嵌是一种非常重要的优化方法，但对于像 Java 编程语言那样的动态的面向对象的编程语言来说，这在传统上一直是非常难以实现的。此外，尽管检测“热点”和内嵌它们所调用的方法已经十分困难，但它仍然还不足以提供全部的 Java 编程语言的语义。这是因为，用 Java 编程语言编写的程序不仅能够“在运行中”改变方法调用的模式，而且能够为一个运行的程序动态地装载新的 Java 代码。

内嵌是基于全局分析的，动态装载使内嵌更加复杂了，因为它改变了一个程序内部的全局关系。一个新的类可能包含了需要被内嵌在适当位置的新的方法。所以，Java HotSpot 性能引擎必须能够动态地逆优化(如果需要，然后再重新优化)先前已经优化过的“热点”，甚至在“热点”代码的执行过程中进行这种操作。没有这种能力，一般的内嵌将不能在基于 Java 的程序上安全地执行。

4．优化编译器

只有性能关键性代码才被编译，这就“购买了时间”，并可将这些时间用于更好的优化。Java HotSpot 性能引擎使用全优化编译器，以此替代了相对简单的 JIT 编译器。全优化编译器可执行所有第一流的优化。例如：死代码删除、循环非变量的提升、普通子表达式删除和连续不断的传送(Constant Propagation)等。它还赋予优化某些特定于 Java 技术的性能。如：空-检查(Null-check)和值域-检查(Range-check)删除等。寄存器分配程序(Register Allocator)是一个用颜色表示分配程序的全局图形，它充分利用了大的寄存器集(Register Sets)。Java HotSpot 性能引擎的全优化编译器的移植性能很强，它依赖相对较小的机器描述文件来描述目标硬件的各个方面。尽管编译器采用了较慢的 JIT 标准，但它仍然比传统的优化编译器要快得多。而且，改善的代码质量也是对由于减少已编译代码的执行次数而节省的时间的一种“回报”。

15.2.2　解释器与编译器

尽管并不是所有的 Java 虚拟机都采用解释器与编译器并存的架构，但许多主流的商用虚拟机，如 HotSpot、J9 等，都同时包含解释器与编译器，解释器与编译器两者各有优势：当程序需要迅速启动和执行的时候，解释器可以首先发挥作用，省去编译的时间，立即执行。当程序运行后，随着时间的推移，编译器逐渐发挥作用，把越来越多的代码编译成本地代码之后，可以获取更高的执行效率。当程序运行环境中内存资源限制较大(如部分嵌入式系统中)，可以使用解释执行节约内存，反之可以使用编译执行来提升效率。同时，解释器还可以作为编译器激进优化时的一个“逃生门”，让编译器根据概率选择一些大多数时候都能提升运行速度的优化手段，当激进优化的假设不成立，如加载了新类后类型继承结构出现变化、出现“罕见陷阱”(Uncommon Trap)时可以通过逆优化(Deoptimization)退回到解释状态继续执行，部分没有解释器的虚拟机中也会采用不进行激进优化的 C1 编译器担任“逃生门”的角色，因此在整个虚拟机执行架构中，解释器与编译器经常是相辅相成地配合工作的，如图 15-1 所示。

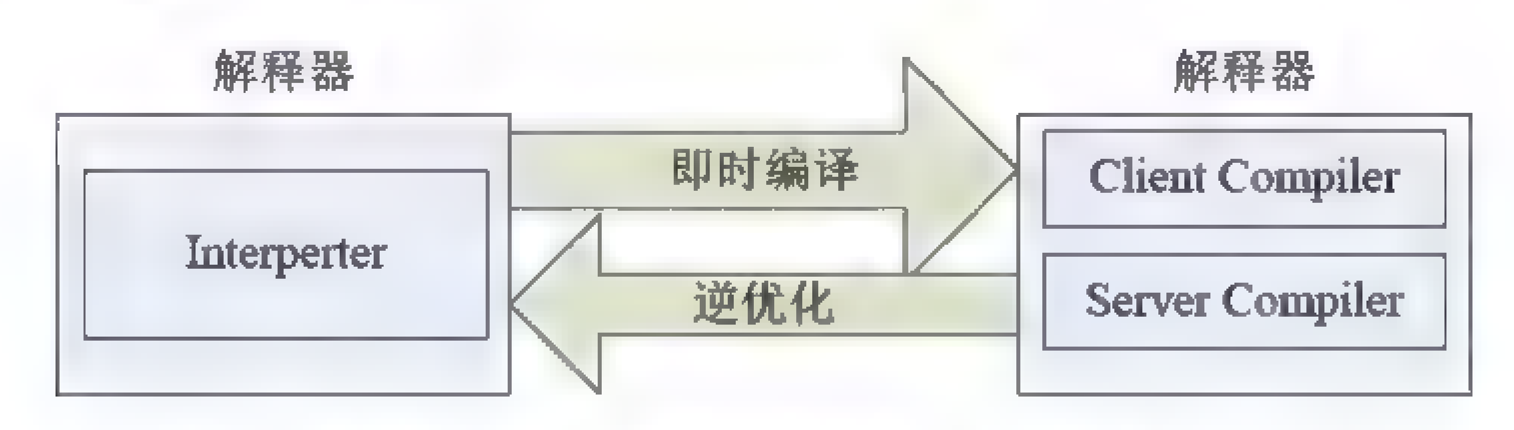


图 15-1　解释器和编译器之间的交互

HotSpot 虚拟机中内置了两个即时编译器，称为 Client Compiler 和 Server Compiler，或者简称为 C1 编译器和 C2 编译器(也叫 Opto 编译器)。目前主流的 HotSpot 虚拟机中(Sun 系列 JDK 1.6 及之前版本的虚拟机)，默认是采用解释器与其中一个编译器直接配合的方式工作，程序使用哪个编译器，取决于虚拟机运行的模式，HotSpot 虚拟机会根据自身版本与宿主机器的硬件性能自动选择运行模式，用户也可以使用-client 或-server 参数去强制指定虚拟机运行在 Client 模式还是 Server 模式。

无论采用的编译器是 Client Compiler 还是 Server Compiler，解释器与编译器搭配使用的方式在虚拟机中被称为“混合模式”(Mixed Mode)，用户可以使用参数-Xint 强制虚拟机运行于“解释模式”(Interpreted Mode)，这时候编译器完全不介入工作，全部代码都使用解释方式执行。另外，也可以使用参数-Xcomp 强制虚拟机运行于“编译模式”(Compiled Mode)，这时候将优先采用编译方式执行程序，但是解释器仍然要在编译无法进行的情况下介入执行过程，可以通过虚拟机的-version 命令的输出结果显示出这三种模式，如代码清单 15-1，请注意加粗字部分。

代码清单 15-1　虚拟机执行模式

```
C:\>java -version
java version "1.6.0 22"
Java (TM) SE Runtime Environment (build 1.6.0 22 b04)
Dynamic Code Evolution 64 Bit Server VM (build 0.2 b02-internal, 19.0 b04-
internal, mixed mode)
```

```
   C:\>java -Xint -version
   java version n1.6.0 22"
   Java(TM) SE Runtime Environment (build 1.6.0 22 b04)
   Dynamic Code Evolution 64-Bit Server VM (build 0.2-b02-internal, 19.0-b04-
internal, intarpreted mode)
   C:\>java -Xcomp -version
   java version "1.6.0 22"
   Java(TM) SE Runtime Environment (build 1.6.0 22-b04)
  Dynamic Code Evolution 64-Bit Server VM (build 0.2-b02-internal, 19.0-b04-
internal, compiled mode)
```

由于即时编译器编译本地代码需要占用程序运行时间，要编译出优化程度更高的代码，所花费的时间可能越长；而且想要编译出优化程度更高的代码，解释器可能还要替编译器收集性能监控信息，这对解释执行的速度也有所影响。为了在程序启动响应速度与运行效率之间达到最佳平衡，HotSpot 虚拟机将会逐渐启用分层编译(Tiered Compilation)的策略，分层编译的概念在 JDK 1.6 时期出现，后来一直处于改进阶段，最终在 JDK 1.7 的 Server 模式虚拟机中作为默认编译策略被开启。分层编译根据编译器编译、优化的规模与耗时，划分出不同的编译层次，其中包括：

- 第 0 层：程序解释执行，解释器不开启性能监控功能(Profiling)，可触发第 1 层编译。
- 第 1 层：也称为 C1 编译，将字节码编译为本地代码，进行简单可靠的优化，如有必要将加入性能监控的逻辑。
- 第 2 层(或 2 层以上)：也称为 C2 编译，也是将字节码编译为本地代码，但是会启用一些编译耗时较长的优化，甚至会根据性能监控信息进行一些不可靠的激进优化。

实施分层编译后，Client Compiler 和 Server Compiler 将会同时工作，许多代码都可能会被多次编译，用 Client Compiler 获取更高的编译速度，用 Server Compiler 来获取更好的编译质量，在解释执行的时候也无需再承担收集性能监控信息的任务。

15.2.3 编译对象与触发条件

在概述中提到过在运行过程中会被即时编译器编译的“热点代码”有两类，即：

- 被多次调用的方法。
- 被多次执行的循环体。

前者很好理解，一个方法被调用得多了，方法体内代码执行的次数自然就多，它成为“热点代码”是理所当然的。而后者则是为了解决当一个方法只被调用过一次或几次，但是方法体内部存在循环次数较多的循环体，这样循环体的代码也被重复执行多次，因此这些代码也应该成为“热点代码”。

对于第一种情况，由于是由方法调用触发的编译，那编译器理所当然地会以整个方法作为编译对象，这种编译也是虚拟机中标准的编译方式。而对于后一种情况，尽管编译动作是由循环体所触发的，但编译器依然会以整个方法(而不是单独的循环体)作为编译对象。这种编译方式因为编译发生在方法执行过程之中，因此被很形象地称为栈上替换(On

Stack Replacement，OSR)。

读者可能还会有疑问，在上面的文字描述里，无论是“多次执行的方法”，还是“多次执行的代码块”，所谓“多次”都不是一个具体、严谨的用语，那到底多少次才算“多次”呢？还有一个问题，就是虚拟机如何统计一个方法或一段代码被执行过多少次呢？解决了这两个问题，也就回答了即时编译被触发的条件。

要知道一段代码是不是热点代码，是不是需要触发即时编译，这个行为称为热点探测(Hot Spot Detection)，其实进行热点探测并不一定要知道方法具体被调用了多少次，目前主要的热点探测判定方式有两种，分别是：

- 基于采样的热点探测(Sample Based Hot Spot Detection)：采用这种方法的虚拟机会周期性地检查各个线程的栈顶，如果发现某个(或某些)方法经常出现在栈顶，那这个方法就是“热点方法”。基于采样的热点探测的好处是实现简单高效，还可以很容易地获取方法调用关系(将调用堆栈展开即可)，缺点是很难精确地确认一个方法的热度，容易因为受到线程阻塞或别的外界因素的影响而扰乱热点探测。
- 基于计数器的热点探测(Counter Based Hot Spot Detection)：采用这种方法的虚拟机会为每个方法(甚至是代码块)建立计数器，统计方法的执行次数，如果执行次数超过限定的阈值就认为它是“热点方法”。这种统计方法实现起来麻烦一些，需要为每个方法建立并维护计数器，而且不能直接获取到方法的调用关系。但是它的统计结果相对来说更加精确、严谨。

在 HotSpot 虚拟机中使用的是第二种——基于计数器的热点探测方法，因此它为每个方法准备了两个计数器：方法调用计数器(Invocation Counter)和回边计数器(Back Edge Counter)。在确定虚拟机运行参数的前提下，这两个计数器都有一个确定的阈值，当计数器超过阈值溢出了，就会触发 JIT 编译。

我们首先来看看方法调用计数器。顾名思义，这个计数器用于统计方法被调用的次数，它的默认阈值在 Client 模式下是 1500 次，在 Server 模式下是 10000 次，这个阈值可以通过虚拟机参数-XX：CompileThreshold 来人工设定。当一个方法被调用时，会先检查该方法是否存在被 JIT 编译过的版本，如果存在，则优先使用编译后的本地代码来执行。如果不存在已被编译过的版本，则将此方法的调用计数器值加 1，然后判断方法调用计数器与回边计数器值之和是否超过方法调用计数器的阈值。如果已超过阈值的话，将会向即时编译器提交一个该方法的代码编译请求。

在默认设置下，执行引擎并不会同步等待编译请求完成，而是继续进入解释器按照解释方式执行字节码，直到提交的请求被编译器编译完成。当编译工作完成之后，这个方法的调用入口地址就会被系统自动改写成新的地址，下一次调用该方法时就会使用已编译的版本。

在默认设置下，方法调用计数器统计的并不是方法被调用的绝对次数，而是一个相对的执行频率，即一段时间之内方法被调用的次数。当超过一定的时间限度，如果方法的调用次数仍然不足以让它提交给即时编译器编译，那这个方法的调用计数器就会被减少一半，这个过程称为方法调用计数器的热度衰减(Counter Decay)，而这段时间就称为此方法统计的半衰周期(Counter Half Life Time)，进行热度衰减的动作是在虚拟机进行垃圾收集时

顺便进行的，可以使用虚拟机参数-XX:-UseCounterDecay 来关闭热度衰减，让方法计数器统计方法调用的绝对次数，这样，只要系统运行时间足够长，绝大部分方法都会被编译成本地代码。另外可以使用-XX:CounterHalfLifeTime 参数设置半衰周期的时间，单位是秒。

现在我们再来看看另外一个计数器——回边计数器，它用于统计一个方法中循环体代码执行的次数，在字节码中遇到控制流向后跳转的指令就称为“回边(Back Edge)”，显然建立回边计数器统计的目的就是为了触发 OSR 编译。

关于回边计数器的阈值，虽然 HotSpot 虚拟机也提供了一个类似于方法调用计数器阈值-XX:CompileThreshold 的参数-XX:BackEdgeThreshold 供用户设置，但准确地说，应当是回边的次数而不是循环次数，因为并非所有的循环都是回边，如空循环实际上就可以视为自己跳转到自己的过程，因此并不算作控制流向后跳转，也不会被回边计数器统计。

是当前的虚拟机实际上并未使用此参数，因此我们需要设置另外一个参数-XX:OnStackReplacePercentage 来间接调整回边计数器的阈值，其计算公式为：

虚拟机运行在 Client 模式下，回边计数器阈值计算公式为：方法调用计数器阈值(CompileThreshold) 乘以 OSR 比率 (OnStackReplacePercentage) 除以 100 。其中 OnStackReplacePercentage 默认值为 933，如果都取默认值，那 Client 模式虚拟机的回边计数器的阈值为 13995。

虚拟机运行在 Server 模式下，回边计数器阈值的计算公式为：方法调用计数器阈值(CompileThreshold) 乘以 (OSR 比率 (OnStackReplacePercentage) 减去解释器监控比率(InterpreterProfilePercentage)的差值)除以 100。其中 OnStackReplacePercentage 默认值为 140，InterpreterProfilePercentage 默认值为 33，如果都取默认值，那么 Server 模式虚拟机回边计数器的阈值为 10700。

当解释器遇到一条回边指令时，会先查找将要执行的代码片段是否有已经编译好的版本，如果有的话，它将会优先执行已编译的代码，否则就把回边计数器的值加 1，然后判断方法调用计数器的值与回边计数器的值两者之和是否超过回边计数器的阈值。当超过阈值的时候，将会提交一个 OSR 编译请求，并且把回边计数器的值降低一些，以便继续在解释器中执行循环，等待编译器输出编译结果。

与方法计数器不同，回边计数器没有计数热度衰减的过程，因此这个计数器统计的就是该方法循环执行的绝对次数。当计数器溢出的时候，它还会把方法计数器的值也调整到溢出状态，这样下次再进入该方法的时候就会执行标准编译过程。对于 Server VM 来说，执行情况会比上面描述的还要复杂。

15.2.4 编译过程

在默认设置下，无论是方法调用产生的即时编译请求，还是 OSR 编译请求，虚拟机在代码编译器还未完成之前，都仍然将按照解释方式继续执行，而编译动作则在后台的编译线程中进行。用户可以通过参数-XX:-BackgroundCompilation 来禁止后台编译，禁止后台编译后，当达到 JIT 的编译条件，执行线程向虚拟机提交编译请求后将会一直等待，直到编译过程完成后再开始执行编译器输出的本地代码。

那在后台执行编译的过程中，编译器做了什么事情呢？Server Compiler 和 Client

Compiler 两个编译器的编译过程是不一样的。对于 Client Compiler 来说，它是一个简单快速的三段式编译器，主要的关注点在于局部性的优化，而放弃了许多耗时较长的全局优化手段。

(1) 在第一个阶段，一个平台独立的前端将字节码构造成一种高级中间代码表示(High-Level Intermediate Representaion，HIR)。HIR 使用静态单分配(Static Single Assignment，SSA)的形式来代表代码值，这可以使得一些在 HIR 的构造过程之中和之后进行的优化动作更容易实现。在此之前编译器会在字节码上完成一部分基础优化，如方法内联、常量传播等优化将会在字节码被构造成 HIR 之前完成。

(2) 在第二个阶段，一个平台相关的后端从 HIR 中产生低级中间代码表示(Low-Level Intermediate Representation，LIR)，而在此之前会在 HIR 上完成另外一些优化，如空值检查消除、范围检查消除等，以便让 HIR 达到更高效的代码表示形式。

(3) 最后的阶段是在平台相关的后端使用线性扫描算法(Linear Scan Register Allocation)在 LIR 上分配寄存器，并在 LIR 上做窥孔(Peephole)优化，然后产生机器代码。而 Server Compiler 则是专门面向服务端的典型应用并为服务端的性能配置特别调整过的编译器，也是一个充分优化过的高级编译器，几乎能达到 GNU C++编译器使用-02 参数时的优化强度，它会执行所有的经典的优化动作，如：无用代码消除(Dead Code Elimination)、循环展开(Loop Unrolling)、循环表达式外提(Loop Expression Hoisting)、公共子表达式消除(Common Subexpression Elimination)、常量传播(Constant Propagation)、基本块重排序(Basic Block Reordering)等，还会实施一些与 Java 语言特性密切相关的优化技术，如范围检查消除(Range Check Elimination)、空值检查消除(Null Check Elimination，不过并非所有的空值检查消除都是依赖编译器进行优化的，有一些是在代码运行过程中自动优化了)等。另外，还可能根据解释器或 Client Compiler 提供的性能监控信息，进行一些不稳定的激进优化，如守护内联(Guarded Inlining)、分支频率预测(Branch Frequency Prediction)等，本章的下半部分将会挑选上述的一部分优化手段进行分析讲解。

Server Compiler 的寄存器分配器是一个全局图着色分配器，它可以充分利用某些处理器架构(如 RISC)上的大寄存器集合。以即时编译的标准来看，Server Compiler 无疑是比较缓慢的，但它的速度仍然远远超过传统的静态优化编译器，而且它相对于 Client Compiler 编译输出的代码质量有所提高，可以减少本地代码的执行时间，从而抵消了额外的编译时间开销，所以也有很多非服务端的应用选择使用 Server 模式的虚拟机运行。

15.2.5　查看与分析即时编译结果

一般来说，虚拟机的即时编译过程对用户程序是完全透明的，虚拟机通过解释执行代码还是编译执行代码，对于用户来说并没有什么影响(执行结果没有影响，速度上会有很大差别)，大多数情况下用户也没有必要知道。但是虚拟机也提供了一些参数用来输出即时编译和某些优化手段(如方法内联)的执行状况，本节将介绍如何从外部观察虚拟机的即时编译行为。

本节中提到的运行参数有一部分需要 Debug 或 FastDebug 版虚拟机的支持，Product 版的虚拟机无法使用这部分参数。如果读者使用的是根据本书第 1 章的教程自己编译的

JDK，请注意将 SKIP DEBUG BUILD 或 SKIP FASTDEBUG BUILD 参数设置为 false，也可以在 OpenjDK 网站上直接下载 FastDebug 版的 JDK。本节中所有的测试都基于代码清单 15-2 所示的 Java 代码。

代码清单 15-2

```
public static final int NUM = 15000;
public static int doubleValue(int i) {
   return i *, 2;
     }
public static long calcSum() {
  long sum = 0;
```

首先运行这段代码，并且确认这段代码是否触发了即时编译，要知道某个方法是否被编译过，可以使用参数-XX:+PrintCompilation 要求虚拟机在即时编译时将被编译成本地代码的方法名称打印出来，如代码清单 15-3 所示(其中带有“%”的输出说明是由回边计数器触发的 OSR 编译)。

代码清单 15-3　被即时编译的代码

```
VM option '+PrintCompilation'
310    1   java .lang. String::charAt(33 bytes)
329    20rg. fenixsoft. jit .Test::calcSum   (26  bytes)
329    30rg. fenixsoft. jit .Test:: doubleValue(4 bytes)
332    1%org.fenixsoft.jit.Test::main@5(20 bytes)
```

从代码清单 15-3 输出的确认信息中可以确认 main(). calcSum()和 doubleValue()方法已经被编译，我们还可以加上参数-XX:+Printlnlining 要求虚拟机输出方法内联信息，如代码清单 15-4。

代码清单 15-4　内联信息

```
VM option '+PrintCompilation'
{
VM option '+PrintInliningi
273    1java.lang.String::charAt (33 bytes)    ;
291    20rg.fenixsoft.jit.Test::calcSum (26 bytes)
@90rg. fenixsoft-j it .Test:: doubleValue     inline( hot)    {
294    30rg. fenixsoft.jit. Test::doubleValue(4 bytes)    l
295    1%org.fenixsoft.jit.Test::main@5  (20 bytes)
@50rg. fenixsoft.jit .Test::calcSum    inline    (hot)    '
@9   0rg. fenixsoft.jit .Test:: doubleValue     inline( hot)    `
kl
```

从代码清单 15-4 的输出中可以看到方法 doubleValue()被内联编译到 calcSum()中，而 calcSum()又被内联编译到方法 main()里面，所以虚拟机再次执行 main()方法的时候(尽管 main()方法并不会运行两次)，calcSum()和 doubleValue()方法都不会再被调用，它们的代码逻辑都被直接内联到 main()方法里面了。

除了查看哪些方法被编译之外，还可以进一步查看即时编译器生成的机器码内容，不过如果虚拟机输出一串 0 和 1，对于我们的阅读来说是没有意义的，机器码必须反汇编成基本的汇编语言才可能被阅读。虚拟机提供了一组通用的反汇编接口，可以接入各种平台下的反汇编适配器来使用，如使用 32 位 x86 平台则选用 hsdis-i386 适配器，其余平台的适

配器还有如 hsdis-amd64、hsdis-sparc 和 hsdis-sparcv9 等，可以下载或自己编译出反汇编适配器后，将其放置在 JRE/bin/client 或/servier 目录下，只要与 jvm.dll 的路径相同即可被虚拟机调用。为虚拟机安装了反汇编适配器之后，就可以使用-XX:+PrintAssembly 参数要求虚拟机打印编译方法的汇编代码了如果没有 hsdis 支持，也可以使用-XX:printOptoAssembly(用于 Server VM)或-XX:+print LIR(用于 Client VM)来输出比较接近最终结果的中间代码表示。

如果除了本地代码的生成结果外，还想再进一步跟踪本地代码生成的具体过程，那还可以使用参数-XX:+PrintCFGToFile(使用 Client Compiler)或-XX:PrintIdealGraphFile(使用 Server Compiler)令虚拟机将编译过程中各个阶段的数据(如对 C1 编译器来说包括：字节码、HIR 生成、LIR 生成、寄存器分配过程、本地代码生成等数据)输出到文件中。然后使用 Java HotSpot Client Compiler Visualizer(使用 Client Compiler)或 Ideal Graph Visualizer(使用 Server Compiler)打开这些数据文件进行分析。

实际使用的时候请注意，要输出 CFG 或 IdealGraph 文件，需要一个 Debug 版的虚拟机支持，Product 版或 FastDebug 版的虚拟机无法输出这些文件。前面提到的使用-XX:+PrintAssembly 参数输出反汇编信息也需要 FastDebug 版的虚拟机才能直接支持，如果使用 Product 版的虚拟机，则需要加入参数-XX:+UnlockDiagnosticVMOptions 打开虚拟机诊断模式后才能使用。

15.3 编译优化技术

Java 程序员都有一个共同的认知，以编译方式执行本地代码比解释方式更快，其中除去虚拟机解释执行字节码时额外消耗的时间以外，还有一个很重要的原因就是 JDK 设计团队几乎把对代码的所有优化措施都集中在了即时编译器之中，所以一般来说即时编译器产生的本地代码会比 Javac 产生的字节码更优秀，接下来我们就介绍一些 HotSpot 虚拟机的即时编译器在生成代码时采用的代码优化技术。

15.3.1 优化技术概览

Sun 官方的 Wiki 上，HotSpot 虚拟机设计团队列出了一个相对比较全面的、即时编译器中采用的优化技术列表，如表 15-1 所示。其中有不少经典编译器的优化手段，也有许多针对 Java 语言(准确地说是针对运行在 Java 虚拟机上的所有语言)本身进行的优化技术，本节将对这些技术进行一遍简单的概览，在后面的几节中，笔者将挑选若干最重要或最典型的优化，与读者一起看看优化前后的代码发生了怎样的变化。

表 15-1　即时编译器优化技术

类　型	优化技术
编译器策略 (compiler tactics)	延迟编译(delayed compilation) 分层编译(tiered compilation) 栈上替换(on-stack replacement) 延迟优化(delayed reoptimization) 程序依赖图表示(program dependence graph representation) 静态单赋值表示(static single assignment representation) 乐观空值断言(optimistic nullness assertions) 乐观类型断言(optimistic type assertions) 乐观类型增强(optimistic type strengthening)
基于性能监控的优化技术 (profile-based techniques)	乐观数组长度增强(optimistic array length strengthening) 裁剪未被选择的分支(untaken branch pruning) 乐观的多态内联(optimistic N-morphic inlining) 分支频率预测(branch frequency prediction) 调用频率预测(call frequency prediction) 精确类型推断(exact type inference) 内存值推断(memory value inference) 内存值跟踪(memory value tracking) 常量折叠(constant folding) 重组(reassociatiQn)
基于证据的优化技术 (proof-based techniques)	操作符退化(operator strength reduction) 空值检查消除(null check elimination) 类型检测退化(type test strength reduction) 类型检测消除(type test elimination) 代数化简(algebraic simplification) 公共子表达式消除(common subexpression elimination)
数据流敏感重写 (flow-sensitive rewrites)	条件常量传播(conditional constant propagation) 基于流承载的类型缩减转换(flow-carried type narrowing) 无用代码消除(dead code elimination) 类型继承关系分析(class hierarchy analysis) 去虚拟机化(devirtualization) 符号常量传播(symbolic constant propagation)
语言相关的优化技术 (language-specific techniques)	自动装箱消除(autobox elimination) 逃逸分析(escape analysis) 锁消除(lock elision) 锁膨胀(lock coarsening) 消除反射(de-reflection) 表达式提升(expression hoisting)

续表

类　型	优化技术
内存及代码位置变换 (memory and placement transformation)	表达式下沉(expression sinking)
	冗余存储消除(redundant store elimination)
	相邻存储合并(adjacent store fusion)
	交汇点分离(merge-point splitting)
	循环展开(loop unrolling)
循环变换(loop transformations)	循环剥离(loop peeling) 安全点消除(safepoint elimination) 迭代范围分离(iteration range splitting) 范围检查消除(range check elimination) 循环向量化(loop vectorization)
全局代码调整 (global code shaping)	内联(inlining) 全局代码外提(global code motion) 基于热度的代码布局(heat-based code layout) Switch 调整(switch balancing)
	本地代码编排(local code scheduling)
	本地代码封包(local code bundling)
	延迟槽填充(delay slot filling)
	着色图寄存器分配(graph-coloring register allocation)
	线性扫描寄存器分配(linear scan register allocation)
控制流图变换 (control flow graph transformation)	复写聚合(copy coalescing)
	常量分裂(constant splitting)
	复写移除(copy removal)
	地址模式匹配(address mode matching)
	指令窥孔优化(instruction peepholing)
	基于确定有限状态机的代码生成(DFA-based code generator)

上述的优化技术看起来很多，而且名字看起来都显得有点“高深莫测”，实际上实现这些优化也许确实有些难度，但大部分技术理解起来都并不困难，笔者举一个最简单的例子来展示其中几种优化技术是如何发挥作用的。首先从原始代码开始，如代码清单 15-5 所示。

代码清单 15-5　优化前的原始代码

```
static class B{
int value.
final int get(){
return value;
)
)
public void foo()  {
y=b. get()j
//…    .do stuff.…
z b. get()j
```

```
sum = y+z;
}
```

首先需要明确一点的是，这些代码的优化变换都是建立在代码的某种中间表示上，绝不是建立在 Java 源码之上的，笔者为了展示方便，使用了 Java 语言的语法来表示这些优化技术所发挥的作用。代码清单 15-5 的代码已经非常简单了，但是仍有许多优化的余地。首先进行方法内联(Method Inlining)，内联的主要目的有两个，一是去除方法调用的成本(如建立栈帧等)，二是为其他优化建立良好的基础。方法内联膨胀之后可以便于在更大范围上进行后续的优化手段，可以获取更好的优化效果，因此各种编译器一般都会把内联优化放在优化序列的靠前位置。内联后的代码如代码清单 15-6 所示。

代码清单 15-6

```
public void foo(){
y = b.value;
z = b.value;
sum = y+z;
}
```

再看消除冗余后的代码(代码清单 15-7)：

代码清单 15-7

```
public void foo(){
y = b.value;
sum = y+y;
}
```

经过优化之后，代码清单 15-7 与代码清单 15-6 所达到的效果是一致的，但是前者比后者省略了许多语句(体现在字节码和机器码指令上的差距会更大)，执行效率也更高。编译器的这些优化技术实现起来也许比较复杂，但是要理解它们的行为对于一个普通的程序员来说是没有困难的，接下来我们继续查看如下的几项优化技术是如何运作的，它们分别是：

- 语言无关的经典优化技术之一：公共子表达式消除。
- 语言相关的经典优化技术之一：数组范围检查消除。
- 最重要的优化技术之一：方法内联。
- 最前沿的优化技术之一：逃逸分析。

15.3.2 公共子表达式消除

公共子表达式消除是一个普遍应用于各种编译器的经典优化技术，它的含义是：如果一个表达式 E 已经被计算过了，并且从先前的计算到现在 E 中所有变量的值都没有发生变化，那么 E 的这次出现就成为了公共子表达式。对于这种表达式，没有必要花时间再对它进行计算，只需要直接用前面计算过的表达式结果代替 E 就可以了。如果这种优化仅限于程序的基本块内，便称为局部公共子表达式消除(Local Common Subexpression Elimination)，如果这种优化的范围涵盖了多个基本块，那就称为全局公共子表达式消除(Global Common Subexpression Elimination)。举个简单的例子来说明它的优化过程，假设存

在如下代码：

```
int d=(c*b)*12+a+(a+b*c);
```

如果这段代码交给 Javac 编译器则不会进行任何优化。当这段代码进入到虚拟机即时编译器后，它将进行如下优化：编译器检测到“c*b”与“b*c”是一样的表达式，而且在计算期间 b 与 c 的值是不变的。因此这条表达式就可能被视为：

```
int d=E*12+a+(a+E);
```

这时候，编译器还可能(取决于哪种虚拟机的编译器及具体的上下文)进行另外一种优化——代数化简(Algebraic Simplification)，把表达式变为：

```
int d=E*13+a*2;
```

表达式进行变换之后，再计算起来就可以节省一些时间了。如果您还对其他的经典编译优化技术感兴趣，可以参考《编译原理》(俗称龙书，推荐使用 Java 的程序员看 2006 年版的紫龙书)中的相关章节。

15.3.3　数组边界检查消除

数组边界检查消除(Array Bounds Checking Elimination)是即时编译器中的一项语言相关的经典优化技术。我们知道 Java 语言是一门动态安全的语言，对数组的读写访问也不像 C、C++那样本质上是裸指针操作。如果有一个数组 foo[]，在 Java 语言中访问数组元素 foo[i]的时候系统将会自动进行上下界的范围检查，即检查 i 必须满足 i>=0&&i< foo.length 的这个条件，否则将抛出一个运行时异常：java.lang.ArrayIndexOutOfBoundsException。这对软件开发者来说是一件很好的事情，即使程序员没有专门编写防御代码，也可以避免大部分的溢出攻击。但是对于虚拟机的执行子系统来说，每次数组元素的读写都带有一次隐含的条件判定操作，对于拥有大量数组访问的程序代码，这无疑也是一种性能负担。

无论如何，为了安全，数组边界检查肯定是必须做的，但数组边界检查是不是必须在运行期间一次不漏地检查则是可以“商量”的事情。例如这个简单的情况：数组下标是一个常量，如 foo[3]，只要在编译期根据数据流分析来确定 foo.length 的值，并判断下标“3”没有越界，执行的时候就无需判断了。更加常见的情况是数组访问发生在循环之中，并且使用循环变量来进行的数组访问，如果编译器只要通过数据流分析就可以判定循环变量的取值范围永远在区间[0，foo.length)之内，那在整个循环中就可以把数组的上下界检查消除掉，这可以节省很多次的条件判断操作。

与语言相关的其他消除操作还有不少，如自动装箱消除(Autobox Elimination)、安全点消除(Safepoint Elimination)、消除反射(Dereflection)等，笔者就不再一一介绍了。

15.3.4　方法内联

在前面的讲解中我们提到过方法内联，它是编译器最重要的优化手段之一，除了消除方法调用的成本之外，它更重要的意义是为其他优化手段建立良好的基础，如代码清单 15-8 所示的简单例子就揭示了内联对其他优化手段的意义：事实上 testInline()方法的内部

全部都是无用的代码，如果不做内联，后续即使进行了无用代码消除的优化，也无法发现任何“Dead Code”，因为如果分开来看，foo()和 testInline()两个方法里的操作都可能有意义。

代码清单 15-8　没有做优化的字节码

```
public static void foo(Object obj){
if ( obj != null {
            System.out.println("do thing");
        }
}
public static void testInline(String [] args){
      Object obj = null;
      Foo (obj);
}
```

方法内联的优化行为看起来很简单，不过是把目标方法的代码“复制”到发起调用的方法之中，避免发生真实的方法调用而已。但实际上 Java 虚拟机中的内联过程远远没有那么简单，因为如果不是即时编译器做了一些特别的努力，按照经典编译原理的优化理论，大多数的 Java 方法都无法进行内联！无法内联的原因是，这是因为只有使用 invokespecial 指令调用的私有方法、实例构造器、父类方法和使用 invokestatic 指令进行调用的静态方法才是在编译期进行解析的，除了上述 4 种方法之外，其他的 Java 方法调用都需要在运行时进行方法接收者的多态选择，并且都有可能存在多于一个版本的方法接收者(最多再除去被 final 修饰的方法这种特殊情况，尽管它使用 invokevirtual 指令调用，但也是非虚方法，Java 语言规范中明确说明了这一点)，简而言之，Java 语言中默认的实例方法就是虚方法。

对于一个虚方法，编译器做内联的时候根本就无法确定应该使用哪个方法版本，例如，前面代码清单 15-7 中把“b.get()”内联为“b.value”，就是不依赖上下文就无法确定 b 的实际类型是什么。假如有 ParentB 和 SubB 两个具有继承关系的类，并且子类重写了父类的 get()方法，那么，是要执行父类的 get()方法还是子类的 get()方法，需要在运行期才能确定，编译期无法得出结论。

由于 Java 语言提倡使用面向对象的编程方式进行编程，而 Java 对象的方法默认就是虚方法，因此 Java 间接鼓励了程序员使用大量的虚方法来完成程序逻辑。根据我们上面的分析，内联与虚方法之间会产生“矛盾”，那该怎么办呢？是不是为了提高执行性能，就要到处使用 final 关键字去修饰方法呢？

为了解决虚方法的内联问题，Java 虚拟机设计团队想了很多办法，首先是引入了一种名为“类型继承关系分析”(Class Hierarchy Analysis，CHA)的技术，这是一种基于整个应用程序的类型分析技术，它用于确定在目前已加载的类中，某个接口是否有多于一种的实现，某个类是否存在子类且子类是否为抽象类等信息。

编译器在进行内联时，如果是非虚方法，那么直接进行内联就可以了，这时候的内联是有稳定前提保障的。如果遇到虚方法，则会向 CHA 查询此方法在当前程序下是否有多个目标版本可供选择，如果查询结果只有一个版本，那也可以进行内联，不过这种内联就属于激进优化，需要预留一个“逃生门”(Guard 条件不成立时的 Slow Path)，称为守护内联(Guarded Inlining)。如果程序的后续执行过程中，虚拟机一直没有加载到会令这个方法

的接收者的继承关系发生变化的类，那这个内联优化的代码就可以一直使用下去。但是如果加载了导致继承关系发生变化的新类，那就需要抛弃掉已经编译的代码，退回到解释状态执行，或者重新进行编译。

如果向 CHA 查询出来的结果是有多个版本的目标方法可供选择，则编译器还将会进行最后一次努力，使用内联缓存(Inline Cache)来完成方法内联，这是一个建立在目标方法正常入口之前的缓存，它的工作原理大致是：在未发生方法调用之前，内联缓存状态为空，当第一次调用发生后，缓存记录下方法接收者的版本信息，并且，每次进行方法调用时都比较接收者版本，如果以后进来的每次调用的方法接收者版本都是一样的，那这个内联还可以一直用下去。如果发生了方法接收者不一致的情况，就说明程序真正使用到了虚方法的多态特性，这时候才会取消内联，查找虚方法表进行方法分派。

所以说，在许多情况下虚拟机进行的内联都是一种激进优化，激进优化的手段在高性能的商用虚拟机中很常见，除了内联之外，对于出现概率很小(通过经验数据或解释器收集到的性能监控信息确定概率大小)的隐式异常、使用概率很小的分支等都可以被激进优化“移除”掉，如果真的出现了小概率事件，这时才会从“逃生门”回到解释状态重新执行。

15.3.5 逃逸分析

逃逸分析(Escape Analysis)是目前 Java 虚拟机中比较前沿的优化技术，它与类型继承关系分析一样，并不是直接优化代码的手段，而是为其他优化手段提供依据的分析技术。

逃逸分析的基本行为就是分析对象动态作用域：当一个对象在方法里面被定义后，它可能被外部方法所引用，例如作为调用参数传递到其他方法中，这种行为称为方法逃逸。甚至还有可能被外部线程访问到，譬如赋值给类变量或可以在其他线程中访问的实例变量，这种行为称为线程逃逸。

如果能证明一个对象不会逃逸到方法或线程之外，也就是别的方法或线程无法通过任何途径访问到这个对象，则可能为这个变量进行一些高效的优化，如：

栈上分配(Stack Allocations)：Java 虚拟机中，在 Java 堆上分配创建对象的内存空间几乎是 Java 程序员都清楚的常识了，Java 堆中的对象对于各个线程都是共享和可见的，只要持有这个对象的引用，就可以访问堆中存储的对象数据。虚拟机的垃圾收集系统可以回收掉堆中不再使用的对象，但回收动作无论是筛选可回收对象，还是回收和整理内存都需要耗费时间。如果确定一个对象不会逃逸出方法之外，那让这个对象在栈上分配内存将会是一个很不错的主意，对象所占用的内存空间就可以随栈帧出栈而销毁。在一般应用中，不会逃逸的局部对象所占的比率很大，如果能使用栈上分配，那大量的对象就会随着方法的结束而自动销毁了，垃圾收集系统的压力将会小很多。

- 同步消除(Synchronization Elimination)：线程同步本身就是一个相对耗时的过程，如果逃逸分析能够确定一个变量不会逃逸出线程，无法被其他线程访问，那这个变量的读写肯定就不会有竞争，对这个变量实施的同步措施也就可以消除掉。
- 标量替换(Scalar Replacement)：标量(Scalar)是指一个数据已经无法再分解成更小的数据来表示了，Java 虚拟机中的原始数据类型(int、long 等数值类型及

reference 类型等)都不能再进一步分解，它们就可以被称为标量。相对的，如果一个数据可以继续分解，那它就被称做聚合量(Aggregate)，Java 中的对象就是最典型的聚合量。如果把一个 Java 对象拆散，根据程序访问的情况，将其使用到的成员变量恢复原始类型来访问就叫做标量替换。如果逃逸分析证明一个对象不会被外部访问，并且这个对象可以被拆散的话，那程序真正执行的时候将可能不创建这个对象，而改为直接创建它的若干个被这个方法使用到的成员变量来代替。将对象拆分后，除了可以让对象的成员变量在栈上(栈上存储的数据，很大机会会被虚拟机分配至物理机器的高速寄存器中存储)分配和读写之外，还可以为后续进一步的优化手段创建条件。

逃逸分析在 Sun JDK 1.6 中实现，但是现在这项优化尚未成熟，仍有巨大的改进余地。不成熟的原因主要是不能保证逃逸分析的性能收益必定高于它的消耗。如果要百分之百准确地判断一个对象是否会逃逸，需要进行数据流敏感的复杂分析，来确定程序各个分支执行时对此对象的影响。这是一个相对高耗时的过程，如果分析完后发现没有几个不逃逸的对象，那时间就白白浪费了，所以目前虚拟机只能采用不那么准确、时间压力相对较小的算法来完成逃逸分析。还有一点是基于逃逸分析的一些优化手段，如前面提到的“栈上分配”，由于 HotSpot 虚拟机目前的实现方式导致栈上分配实现起来比较复杂，因此在 HotSpot 中暂时还没有做这项优化。

在测试结果上，实施逃逸分析后的程序在 MicroBenchmarks 中往往能运行出不错的成绩，但是在实际的应用程序，尤其是大型程序中，反而发现实施逃逸分析可能会出现效果不稳定的情况，或因分析过程耗时却无法有效地判别出非逃逸对象而导致性能(即时编译的收益)有所下降，所以即使是 Server Compiler，也默认不开启逃逸分析，甚至在某些版本(如 JDK 1.6 Update 18)中还曾经短暂地完全禁止了这项优化。

如果有需要，并且确认对程序运行有益，用户可以使用参数-XX:+DoEscapeAnalysis 来手动开启逃逸分析，开启之后可以通过参数-XX:+PrintEscapeAnalysis 来查看分析结果。另外，用户可以使用参数-XX:+EliminateAllocations 来开启标量替换，使用参数+XX:+EliminateLocks 来开启同步消除，使用参数-XX:+PrintEliminateAllocations 来查看标量的替换情况。

尽管目前逃逸分析的技术仍未完全成熟，它却是即时编译器优化技术一个重要的发展方向，在日后的虚拟机中，逃逸分析技术肯定会支撑起一系列实用有效的优化技术。

15.4 Java 与 C/C++的编译器对比

大多数程序员都认为 C/C++会比 Java 语言快，甚至觉得从 Java 语言诞生以来，“执行速度缓慢”的帽子就应当被扣在头顶，这种观点的出现是由于 Java 刚出现的时候即时初稿完成之前，在最新的 JDK 1.6 Update 23 的 Server Compiler 中已默认开启了逃逸分析。

编译技术还不成熟，主要靠解释器执行的 Java 语言性能确实比较低下。但是在今天即时编译技术已经发展成熟，Java 语言有可能在速度上与 C/C++ 一争高下吗？要想知道这个问题的答案得从两者的编译器谈起。

Java 与 C/C++的编译器对比实际上代表了最经典的即时编译器与静态编译器的对比，很大程度上也决定了 Java 与 C/C++的性能对比的结果，因为无论是 C/C++还是 Java 代码，最终编译之后被机器执行的都是本地机器码，哪种语言的性能更高，除了它们自身的 API 库实现得好坏以外，其余的比较就成了一场“拼编译器”和“拼输出代码质量”的游戏。当然，这种比较也是剔除了开发效率的片面对比，语言间孰优孰劣、谁快谁慢的问题都是很难有结果的争论，下面我们就回到正题，看看这两种语言的编译器各有何种优势。

Java 虚拟机的即时编译器与 C/C++的静态优化编译器相比，可能会由于下列这些原因导致输出的本地代码有一些劣势(下面列举的也包括一些虚拟机执行子系统的性能劣势)：

(1) 因为即时编译器运行占用的是用户程序的运行时间，具有很大的时间压力，它能提供的优化手段也严重受制于编译成本。如果编译速度不能达到要求，那用户将在启动程序或程序的某部分察觉到重大延迟，这点使得即时编译器不敢随便引入大规模的优化技术，而编译的时间成本在静态优化编译器中并不是主要的关注点。

(2) Java 语言是动态的类型安全语言，这就意味着需要由虚拟机来确保程序不会违反语言的语义或访问非结构化内存。在实现层面上看，这就意味着虚拟机必须频繁地进行动态检查，如实例方法访问时检查空指针、数组元素访问时检查上下界范围、类型转换时检查继承关系，等等。对于这类程序代码没有明确写出的检查行为，尽管编译器会努力进行优化，但是总体上仍然要消耗不少的运行时间。

(3) Java 语言中虽然没有 virutal 关键字，但是使用虚方法的频率却远远大于 C/C++语言，这就意味着运行时对方法接收者进行多态选择的频率要远远大于 C/C++语言，也意味着即时编译器在进行一些优化(如前面提到的方法内联)时的难度要远远大于 C/C++的静态优化编译器。

(4) Java 语言是可以动态扩展的语言，运行时加载新的类可能改变程序类型的继承关系，这使得很多全局的优化都难以进行，因为编译器无法看见程序的全貌，许多全局的优化措施都只能以激进优化的方式来完成，编译器不得不时刻注意并随着类型的变化而在运行时撤销或重新进行一些优化。

(5) Java 语言中对象的内存分配都是在堆上进行的，只有方法中的局部变量才能在栈上分配。而 C/C++的对象则有多种内存分配方式，既可能在堆上分配，又可能在栈上分配，如果可以在栈上分配线程私有的对象，将减轻内存回收的压力。另外，C/C++中主要由用户程序代码来回收分配的内存，这就不存在无用对象筛选的过程，因此效率上(仅指运行效率，排除了开发效率)也比垃圾收集机制要高。

上面说了一大堆 Java 语言相对 C/C++的劣势，倒不是说 Java 就真的不如 C/C++，相信读者也注意到了，Java 语言的这些性能上的劣势都是为了换取开发效率上的优势而付出的代价，动态安全、动态扩展、垃圾回收这些“拖后腿”的特性都为 Java 语言的开发效率作出了很大的贡献。何况，还有 Java 的即时编译器能做，而 C/C++的静态优化编译器不能做的优化：由于 C/C++编译器的静态特性，以运行期性能监控为基础的优化措施它都无法进行，如调用频率预测(Call Frequency Prediction)、分支频率预测(Branch Frequency Prediction)、裁剪未被选择的分支(Untaken Branch Pruning)等，这些都会成为 Java 语言独有的性能优势。

Java

第 16 章

内存模型和线程

并发处理的广泛应用是使得 Amdahl 定律代替摩尔定律成为计算机性能发展源动力的根本原因，也是人类压榨计算机运算能力最有力的武器。本章将详细讲解 JVM 内存模型和线程的基本知识，介绍虚拟机如何实现多线程、多线程之间由于共享和竞争数据而导致的一系列问题及解决方案，为读者学习后面的知识打下基础。

16.1 Java 的多线程

线程是一个程序内部的顺序控制流。一个进程相当于一个任务，一个线程相当于一个任务中的一条执行路径。多进程在操作系统中能同时运行多个任务(程序)；在同一个应用程序中有多个顺序流同时执行。Java 的线程是通过 java.lang.Thread 类来实现的；JVM 启动时会有一个由主方法(public static void main(){})所定义的线程；可以通过创建 Thread 的实例来创建新的线程；每个线程都是通过某个特定 Thread 对象所对应的方法 run()来完成其操作的，方法 run()称为线程体，通过调用 Thread 类的 start()方法来启动一个线程。

多任务处理在现代计算机操作系统中几乎已是一项必备的功能了。在许多情况下，让计算机同时去做几件事情，不仅是因为计算机的运算能力强大，还有一个很重要的原因是计算机的运算速度与它的存储和通信子系统速度的差距太大，大部分时间都花在了磁盘 I/O、网络通信和数据库访问上。如果不希望处理器在大部分时间里都处于等待其他资源的状态，就必须使用一些手段去把处理器的运算能力“压榨”出来，否则就会造成很大的“浪费”，而让计算机同时处理几项任务则是最容易想到、也被证明是非常有效的“压榨”手段。

除了充分利用计算机处理器的能力外，一个服务端同时对多个客户端提供服务则是另一个更具体的并发应用场景。衡量一个服务性能的高低好坏，每秒事务处理数(Transactions Per Second，TPS)是最重要的指标之一，它代表着一秒内服务端平均能响应的请求总数，而 TPS 值与程序的并发能力又有非常密切的关系。对于计算量相同的任务，程序线程并发协调得越有条不紊，效率自然就会越高；反之，线程之间频繁阻塞甚至死锁，将会大大降低程序的并发能力。

服务端是 Java 语言最擅长的领域之一，这个领域的应用占了 Java 应用中最大的一块份额，不过如何写好并发应用程序却是程序开发的难点之一，处理好并发方面的问题通常需要更多的经验。幸好 Java 语言和虚拟机提供了许多工具，把并发编程的门槛降低了不少。另外，各种中间件服务器、各类框架都努力地替程序员处理尽可能多的线程并发细节，使得程序员在编码时能更关注业务逻辑，而不是花费大部分时间去关注此服务会同时被多少人调用。但是无论语言、中间件和框架如何先进，我们都不能期望它们能独立完成并发处理的所有事情，了解并发的内幕也是成为一个高级程序员不可缺少的课程。

一般来说,我们把正在计算机中执行的程序叫做“进程”(Process) ,而不将其称为程序(Program)。所谓“线程”(Thread)，是“进程”中某个单一顺序的控制流。新兴的操作系统，如 Mac、Windows NT、Windows 95 等大多采用多线程的概念，把线程视为基本执行单位，线程也是 Java 中的相当重要的组成部分之一。

甚至最简单的 Applet 也是由多个线程来完成的。在 Java 中，任何一个 Applet 的 paint()和 update()方法都是由 AWT(Abstract Window Toolkit)绘图与事件处理线程调用的，而 Applet 主要的里程碑方法——init()、start()、stop()和 destory()是由执行该 Applet 的应用调用的。

单线程的概念没有什么新的地方，真正有趣的是在一个程序中同时使用多个线程来完成不同的任务。某些地方用轻量进程(Lightweig ht Process)来代替线程,线程与真正进程的相似性在于它们都是单一顺序控制流。然而线程被认为轻量是由于它运行于整个程序的上下文内，能使用整个程序共有的资源和程序环境。

作为单一顺序控制流，在运行的程序内线程必须拥有一些资源作为必要的开销。例如必须有执行堆栈和程序计数器在线程内执行的代码只在它的上下文中起作用，因此某些地方用“执行上下文”来代替“线程”。

16.2　硬件的效率与一致性

在正式讲解 Java 虚拟机并发相关的知识之前，我们先花费一点时间去了解一下物理计算机中的并发问题，物理机遇到的并发问题与虚拟机中的情况有不少相似之处，物理机对并发的处理方案对虚拟机的实现也有相当大的参考意义。

“让计算机并发执行若干个运算任务”与“更充分地利用计算机处理器的效能”之间的因果关系，看起来顺理成章，实际上并没有想象中的那么容易实现，因为所有的运算任务都不可能只靠处理器“计算”就能完成，至少与内存的交互，如读取运算数据、存储运算结果等，就是很难消除的(不能仅仅靠寄存器来解决)。由于计算机的存储设备与处理器的运算速度之间有着几个数量级的差距，所以现代计算机系统都不得不加入一层读写速度尽可能接近处理器运算速度的高速缓存(Cache)来作为内存与处理器之间的缓冲：将运算需要使用到的数据复制到缓存中，让运算能快速进行，当运算结束后再从缓存同步回内存之中，这样处理器就无须等待缓慢的内存读写了。

基于高速缓存的存储交互很好地解决了处理器与内存的速度矛盾，但是也引入了新的问题：缓存一致性(Cache Coherence)。在多处理器系统中，每个处理器都有自己的高速缓存，而它们又共享同一主内存(Main Memory)。当多个处理器的运算任务都涉及同一块主内存区域时，将可能导致各自的缓存数据不一致的情况，如果真的发生这种情况，那同步回到主内存时以谁的缓存数据为准呢？为了解决一致性的问题，需要各个处理器访问缓存时都遵循一些协议，在读写时要根据协议来进行操作，这类协议有 MSI、MESI (Illinois Protocol)、MOSI、Synapse、Firefly 及 Dragon Protocol，等等。Java 虚拟机内存模型中定义的内存访问操作与硬件的缓存访问操作是具有可比性的。

除此之外，为了使得处理器内部的运算单元能尽量被充分利用，处理器可能会对输入代码进行乱序执行(Out-Of-Order Execution)优化，处理器会在计算之后将乱序执行的结果重组，保证该结果与顺序执行的结果是一致的，但并不保证程序中各个语句计算的先后顺序与输入代码中的顺序一致，因此如果存在一个计算任务依赖另外一个计算任务的中间结果，那么其顺序性并不能靠代码的先后顺序来保证。与处理器的乱序执行优化类似，Java 虚拟机的即时编译器中也有类似的指令重排序(Instruction Reorder)优化。

16.3　Java 内存模型

不同的平台，内存模型是不一样的，但是 JVM 的内存模型规范是统一的。其实 Java 的多线程并发问题最终都会反映在 Java 的内存模型上，所谓线程安全无非是要控制多个线程对某个资源的有序访问或修改。总结 Java 的内存模型，要解决两个主要的问题：可见性和有序性。

我们都知道计算机有高速缓存的存在，处理器并不是每次处理数据都是取内存的。JVM 定义了自己的内存模型，屏蔽了底层平台内存管理细节，对于 Java 开发人员，要清楚在 JVM 内存模型的基础上，如果解决多线程的可见性和有序性。

那么，何谓可见性？多个线程之间是不能互相传递数据通信的，它们之间的沟通只能通过共享变量来进行。Java 内存模型(JMM)规定了 JVM 有主内存，主内存是多个线程共享的。当新建一个对象的时候，也是被分配在主内存中，每个线程都有自己的工作内存，工作内存存储了主存的某些对象的副本，当然线程的工作内存大小是有限制的。当线程操作某个对象时，执行顺序如下：

(1) 从主存复制变量到当前工作内存(read and load)。

(2) 执行代码，改变共享变量值(use and assign)。

(3) 用工作内存数据刷新主存相关内容(store and write)。

16.3.1　Java 内存模型概述

Java 平台自动集成了线程 以及多处理器技术 ，这种集成程度比 Java 以前诞生的计算机语言要厉害很多，该语言针对多种异构平台的平台独立性，而使用的多线程技术支持也是具有开拓性的一面，有时候在开发 Java 同步和线程安全要求很严格的程序时，往往容易混淆的一个概念就是内存模型。究竟什么是内存模型？内存模型描述了程序中各个变量(实例域、静态域和数组元素)之间的关系，以及在实际计算机系统中将变量存储到内存和从内存中取出变量这样的底层细节，对象最终是存储在内存里面的，这点没有错，但是编译器、运行库、处理器或者系统缓存可以有特权在变量指定内存位置存储或者取出变量的值。

JVM 规范定义了线程对主存的操作指令：read、load、use、assign、store、write。当一个共享变量在多个线程的工作内存中都有副本时，如果一个线程修改了这个共享变量，那么其他线程应该能够看到这个被修改后的值，这就是多线程的可见性问题。那么，什么是有序性呢？线程在引用变量时不能直接从主内存中引用，如果线程工作内存中没有该变量,则会从主内存中拷贝一个副本到工作内存中，这个过程为 read-load，完成后线程会引用该副本。当同一线程再度引用该字段时,有可能重新从主存中获取变量副本(read-load-use),也有可能直接引用原来的副本(use)，也就是说 read、load、use 的顺序可以由 JVM 实现系统决定。

线程不能直接为主存中中字段赋值，它会将值指定给工作内存中的变量副本(assign)，

完成后这个变量副本会同步到主存储区(store-write)，至于何时同步过去，根据 JVM 实现系统决定，有该字段则会从主内存中将该字段赋值到工作内存中，这个过程为 read-load，完成后线程会引用该变量副本，当同一线程多次重复对字段赋值时，比如：

```
for(int i=0;i<10;i++)
  a++;
```

线程有可能只对工作内存中的副本进行赋值，只到最后一次赋值后才同步到主存储区，所以 assign,store,weite 顺序可以由 JVM 实现系统决定。假设有一个共享变量 x，线程 a 执行 x=x+1。从上面的描述中可以知道 x=x+1 并不是一个原子操作，它的执行过程如下：

(1) 从主存中读取变量 x 副本到工作内存。

(2) 给 x 加 1。

(3) 将 x 加 1 后的值写回主存。

如果另外一个线程 b 执行 x=x-1，执行过程如下：

(1) 从主存中读取变量 x 副本到工作内存。

(2) x 减 1。

(3) 将 x 减 1 后的值写回主存。

那么显然，最终的 x 的值是不可靠的。假设 x 现在为 10，线程 a 加 1，线程 b 减 1，从表面上看，似乎最终 x 还是为 10，但是多线程情况下会有这种情况发生：

- 线程 a 从主存读取 x 副本到工作内存，工作内存中 x 值为 10。
- 线程 b 从主存读取 x 副本到工作内存，工作内存中 x 值为 10。
- 线程 a 将工作内存中 x 加 1，工作内存中 x 值为 11。
- 线程 a 将 x 提交主存中，主存中 x 为 11。
- 线程 b 将工作内存中 x 值减 1，工作内存中 x 值为 9。
- 线程 b 将 x 提交到中主存中，主存中 x 为 9。

同样，x 有可能为 11，如果 x 是一个银行账户，线程 a 存款，线程 b 扣款，显然这样是有严重问题的，要解决这个问题，必须保证线程 a 和线程 b 是有序执行的，并且每个线程执行的加 1 或减 1 是一个原子操作。看看下面的代码：

```
public class Account {

    private int balance;

    public Account(int balance) {
        this.balance = balance;
    }

    public int getBalance() {
        return balance;
    }

    public void add(int num) {
        balance = balance + num;
    }
```

```
    public void withdraw(int num) {
        balance = balance - num;
    }

    public static void main(String[] args) throws InterruptedException {
        Account account = new Account(1000);
        Thread a = new Thread(new AddThread(account, 20), "add");
        Thread b = new Thread(new WithdrawThread(account, 20), "withdraw");
        a.start();
        b.start();
        a.join();
        b.join();
        System.out.println(account.getBalance());
    }

    static class AddThread implements Runnable {
        Account account;
        int     amount;

        public AddThread(Account account, int amount) {
            this.account = account;
            this.amount = amount;
        }

        public void run() {
            for (int i = 0; i < 200000; i++) {
                account.add(amount);
            }
        }
    }

    static class WithdrawThread implements Runnable {
        Account account;
        int     amount;

        public WithdrawThread(Account account, int amount) {
            this.account = account;
            this.amount = amount;
        }

        public void run() {
            for (int i = 0; i < 100000; i++) {
                account.withdraw(amount);
            }
        }
    }
}
```

第一次执行结果为 10200，第二次执行结果为 1060，每次执行的结果都是不确定的，因为线程的执行顺序是不可预见的。这是 Java 同步产生的根源，synchronized 关键字保证了多个线程对于同步块是互斥的，synchronized 作为一种同步手段，解决 Java 多线程的执行有序性和内存可见性，而 volatile 关键字之解决多线程的内存可见性问题。

16.3.2 主内存与工作内存

Java 内存模型的主要目标是定义程序中各个变量的访问规则，即在虚拟机中将变量存储到内存和从内存中取出变量这样的底层细节。此处的变量(Variable)与 Java 编程中所说的变量略有区别，它包括了实例字段、静态字段和构成数组对象的元素，但是不包括局部变量与方法参数，因为后者是线程私有的，不会被共享，自然就不存在竞争问题。为了获得较好的执行效能，Java 内存模型并没有限制执行引擎使用处理器的特定寄存器或缓存来和主内存进行交互，也没有限制即时编译器调整代码执行顺序这类权利。

Java 内存模型规定了所有的变量都存储在主内存(Main Memory)中(此处的主内存与介绍物理硬件时的主内存名字一样，两者也可以互相类比，但此处仅是虚拟机内存的一部分)。每条线程还有自己的工作内存(Working Memory，可与前面所讲的处理器高速缓存类比)，线程的工作内存中保存了被该线程使用到的变量的主内存副本拷贝，线程对变量的所有操作(读取、赋值等)都必须在工作内存中进行，而不能直接读写主内存中的变量。不同的线程之间也无法直接访问对方工作内存中的变量，线程间变量值的传递均需要通过主内存来完成，线程、主内存、工作内存三者的交互关系如图 16-1 所示。

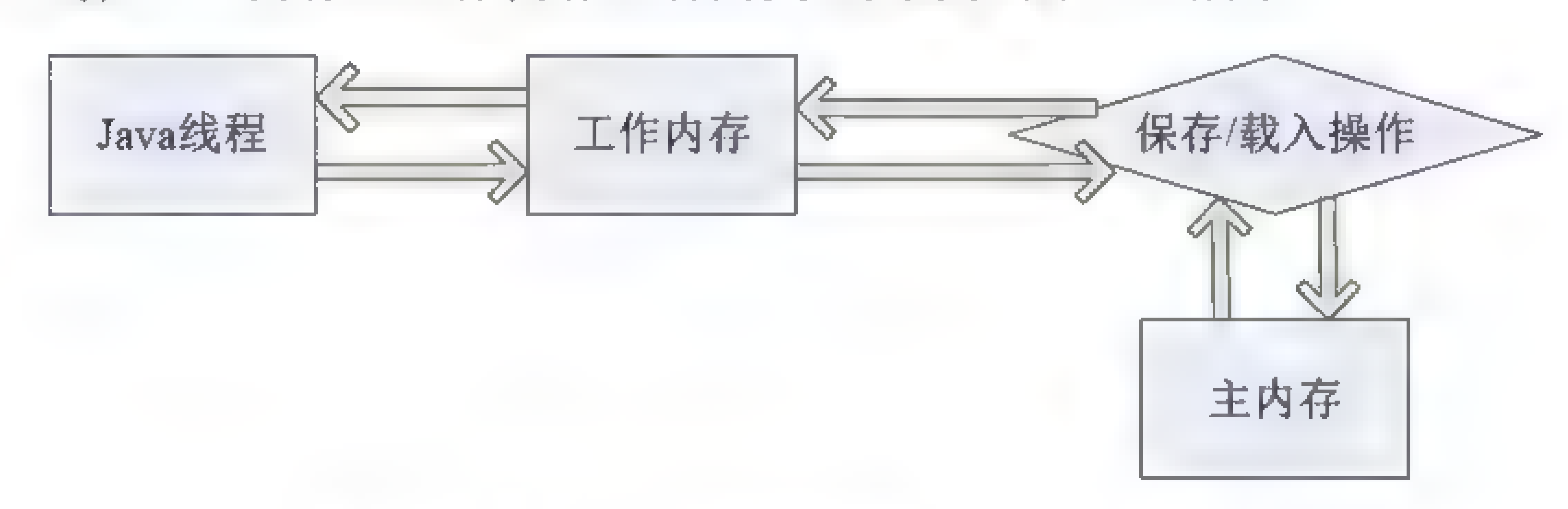


图 16-1 线程、主内存和工作内存之间的交互关系

这里所讲的主内存、工作内存与本书第 2 章所讲的 Java 内存区域中的 Java 堆、栈、方法区等并不是同一个层次的内存划分。如果两者一定要勉强对应起来，那从变量、主内存、工作内存的定义来看，主内存主要对应于 Java 堆中对象的实例数据部分，而工作内存则对应于虚拟机栈中的部分区域。从更低的层次来说，主内存就是硬件的内存，而为了获取更好的运行速度，虚拟机及硬件系统可能会让工作内存优先存储于寄存器和高速缓存中。

16.3.3 内存间交互操作

关于主内存与工作内存之间具体的交互协议，即一个变量如何从主内存拷贝到工作内存、如何从工作内存同步回主内存之类的实现细节，Java 内存模型中定义了以下 8 种操作来完成@：

- lock(锁定)：作用于主内存的变量，它把一个变量标识为一条线程独占的状态。
- unlock(解锁)：作用于主内存的变量，它把一个处于锁定状态的变量释放出来，释放后的变量才可以被其他线程锁定。
- read(读取)：作用于主内存的变量，它把一个变量的值从主内存传输到线程的工作

内存中，以便随后的 load 动作使用。

- load(载入)：作用于工作内存的变量，它把 read 操作从主内存中得到的变量值放入工作内存的变量副本中。
- use(使用)：作用于工作内存的变量，它把工作内存中一个变量的值传递给执行引擎，每当虚拟机遇到一个需要使用到变量的值的字节码指令时将会执行这个操作。
- assign(赋值)：作用于工作内存的变量，它把一个从执行引擎接收到的值赋值给工作内存的变量，每当虚拟机遇到一个给变量赋值的字节码指令时执行这个操作。
- store(存储)：作用于工作内存的变量，它把工作内存中一个变量的值传送到主内存中，以便随后的 write 操作使用。
- write(写入)：作用于主内存的变量，它把 store 操作从工作内存中得到的变量的值放入主内存的变量中。

如果要把一个变量从主内存复制到工作内存，那就要按顺序地执行 read 和 load 操作，如果要把变量从工作内存同步回主内存，就要按顺序地执行 store 和 write 操作。注意，Java 内存模型只要求上述两个操作必须按顺序执行，而没有保证必须是连续执行。也就是说 read 与 load 之间、store 与 write 之间是可插入其他指令的，如对主内存中的变量 a、b 进行访问时，一种可能出现的顺序是 read a、read b、load b、load a。除此之外，Java 内存模型还规定了在执行上述 8 种基本操作时必须满足如下规则：

- 不允许 read 和 load、store 和 write 操作之一单独出现，即不允许一个变量从主内存读取了但工作内存不接受，或者从工作内存发起回写了但主内存不接受的情况出现。
- 不允许一个线程丢弃它的最近的 assign 操作，即变量在工作内存中改变了之后必须把该变化同步回主内存。
- 不允许一个线程无原因地(没有发生过任何 assign 操作)把数据从线程的工作内存同步回主内存中。
- 一个新的变量只能在主内存中“诞生”，不允许在工作内存中直接使用一个未被初始化(load 或 assign)的变量，换句话说就是对一个变量实施 use 和 store 操作之前，必须先执行过了 assign 和 load 操作。
- 一个变量在同一个时刻只允许一条线程对其进行 lock 操作，但 lock 操作可以被同一条线程重复执行多次，多次执行 lock 后，只有执行相同次数的 unlock 操作，变量才会被解锁。
- 如果对一个变量执行 lock 操作，将会清空工作内存中此变量的值，在执行引擎使用这个变量前，需要重新执行 load 或 assign 操作初始化变量的值。
- 如果一个变量事先没有被 lock 操作锁定，则不允许对它执行 unlock 操作；也不允许去 unlock 一个被其他线程锁定住的变量。
- 对一个变量执行 unlock 操作之前，必须先把此变量同步回主内存中(执行 store 和 write 操作)。

16.3.4　volatile 型变量

如果一个变量声明为 volatile 类型，那么每个线程对该变量实施的动作有以下附加的规则，假定 T 表示一个线程，V，W 表示 volatile 类型变量。

(1) 只有当线程 T 对变量 V 执行的前一个动作是 load 的时候，线程 T 才能对变量 V 执行 use 动作；并且，只有当线程 T 对变量 V 执行的后一个动作是 use 的时候，线程 T 才能对变量 V 执行 load 动作。线程 T 对就是 V 的 use 动作可以认为是和线程 T 对变量 V 的 load 动作相应的 read 动作相关联(这样可以保证看其他线程对变量 V 所做的修改后的值，即使用时先去从主内存中加载)。

(2) 只有当线程 T 对变量 V 执行的前一个动作是 assign 的时候，线程 T 才能对变量 V 执行 store 动作；并且，只有当线程 T 对变量 V 执行的后一个动作是 store 的时候，线程 T 才能对变量 V 执行 assign 动作。线程 T 对就是 V 的 assign 动作可以认为是和线程 T 对变量 V 的 store 动作相应的 write 动作相关联(这样可以保证其他线程可以看到自己对变量 V 所做的修改，即修改后写回主内存中)。

(3) 假定动作 A 是线程 T 对变量 V 实施的 use 或 assign 动作，假定动作 F 是和动作 A 相关联的 load 或 store 动作，假定动作 P 是和动作 F 相应的对变量 V 的 read 或 write 动作；类似的，假定动作 B 是线程 T 对变量 W 实施的 use 或 assign 动作，假定动作 G 是和动作 B 相关联的 load 或 store 动作，假定动作 Q 是和动作 G 相应的对变量 W 的 read 或 write 动作。如果 A 先于 B，那么 P 先于 Q(不严格的：为了一个线程 T，主内存实施对给定的 volatile 变量的主拷贝的动作必须遵循和线程执行时要求的一样的先后顺序。也即将 V、W 变量写回到主内存的顺序与程序代码行对 V、W 赋值先后顺序一样；线程将 V、W 变量从主内存读取出来的顺序与程序代码行对 V、W 使用先后顺序一样。即 volate 禁止了变量间的重新排序问题)。该规则进一步加强了多线程访问共享变量的安全性，这条规则是针对多线程提出的。

对声明为 volatile 的变量的规则有效地保证了：线程对一个声明为 volatile 的变量的每个 use 或 assign 动作只要访问主内存一次，并且依照线程的执行语义所指定的次序访问主内存，然而，对没有声明为 volatile 的变量的 read 或 write 动作，这样的内存动作是没有次序限制的。

volatile 的变量除了具有可见性外，还禁止了多个变量间的 Reordering。

请看下面的代码：

```
class Sample{
    int a=1,b=2;
    void hither(){
        a=b;
    }

    void yon(){
        b=a;
    }
}
```

让我们考虑调用 hither 的线程，按照规则，该线程必须执行变量 b 的 use 动作，在它后面要执行变量 a 的 assign 动作，这是对 hither 的最低要求(即同一线程内一定是按照程序语义顺序来执行)。

现在线程对变量 b 的第一个动作不能为 use，但是可以为 assign 或 load。这里对 b 的一个 assign 动作不可能发生，因为这里根本就没有赋值调用，所以这里只有对变量 b 的 load 动作。而线程对这个 load 动作必须有一个更早的主内存对变量 b 的 read 动作。

在对变量 a 进行 assign 动作后，线程可选地(因为没有使用同步)存储变量 a 的值，如果线程要存储这个值，那么线程实施 store 动作，并且主内存接着实施变量 a 的 write 动作。调用方法 yon 的线程的情况是类似的，只是 a 和 b 交换了各自的角色。

假定 ha 和 hb 是调用 hither 的线程的变量 a 和 b 的工作拷贝，ya 和 yb 是调用 yon 线程的变量 a 和 b 的工作拷贝，ma 和 mb 是主内存中变量 a 和变量 b 的主拷贝，初始化 ma=1，mb=2，下面是动作的可能结果：

(1) ha=2，hb=2，ya=2，yb=2，ma=2，mb=2(结果是 b 复制给了 a)。

(2) ha=1，hb=1，ya=1，yb=1，ma=1，mb=1(结果是 a 复制给了 b)。

(3) ha=2，hb=2，ya=1，yb=1，ma=2，mb=1(结果是 a、b 交换了)。

使用以下程序进行测试：

```
class Sample {
    /*
     * 不管 a,b 是否使用 volatile 修饰，都会出现 a、b 值交换。因为 a=b、b=a 并不是原子性
     * 的，因为这两条语句都会涉及使用与赋值两个动作，完全有可能在访问操作后切换到
     * 另一线程，而 volatile 并不像 synchronized 那样具有原子特性
     */
    volatile int a = 1;
    volatile int b = 2;
    void hither() {
        a = b;
    }
    synchronized void yon() {
        b = a;
    }
}
public class Test {
    public static void main(String[] args) throws Exception {
        while (!Thread.currentThread().isInterrupted()) {
            final Sample s = new Sample();
            final Thread hither = new Thread() {
                public void run() {
                    s.hither();
                }
            };
            final Thread yon = new Thread() {
                public void run() {
                    s.yon();
                }
            };
            hither.start();
            yon.start();
```

```
            new Thread() {
                public void run() {
                    try {
                        hither.join();
                        yon.join();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    if (s.a != s.b) {
                // 某次打印结果 Thread-332984: a=2 b=1
                System.out.println(this.getName() + ": a=" + s.a + " b=" + s.b);
                        System.exit(0);
                    }
                }
            }.start();
            Thread.yield();
        }
    }
}
```

上面使用 volatile 同时修改这两个变量还是不行的，除非两个方法同时(注：只一个方法使用也是不管用的)使用 synchronized：

```
class Sample {
    int a = 1;
    int b = 2;
    synchronized void hither() {
        a = b;
    }
    synchronized void yon() {
        b = a;
    }
}
```

lock 和 unlock 动作对主内存的动作次序提出了更多的限制。在一个线程的 lock 动作和 unlock 动作之间，另一个线程不能实施 lock 动作，而且，unlock 动作前需要实施 store 动作和 write 动作，下面是仅可能发现的顺序，从结果看出要么是 a，要么 是 b，不可能出现两都交换的情况：

(1) ha=2，hb=2，ya=2，yb=2，ma=2，mb=2(结果是 b 复制给了 a)。

(2) ha=1，hb=1，ya=1，yb=1，ma=1，mb=1(结果是 a 复制给了 b)。

由此可见，volatile 字段被用来在线程之间 Communicate State(交流规则)。任意线程所 read 的 volatile 字段的值都是最新的。原因有以下有 4 点：

(1) 编译器和 JVM 会阻止将 volatile 字段的值放入处理器寄存器(Register)；

(2) 在 write volatile 字段之后，其值会被 flush 出处理器 cache，写回 memory；

(3) 在 read volatile 字段之前，会 invalidate(验证)处理器 cache。因此，上述两条便保证了每次 read 的值都是 memory 中的，即具有“可见性”这一特性。

(4) 禁止 reorder(重排序，即与原程序指定的顺序不一致)任意两个 volatile 变量，并且同时严格限制(尽管没有禁止)reorder volatile 变量周围的非 volatile 变量。这一点即 volatile 具有变量的“顺序性”，即指令不会重新排序，而是按照程序指定的顺序执行。

注意：在旧的内存模型下，对 volatile 修改的变量的访问顺序不能进行重新排序，但可以对非 volatile 变量进行排序，但这样又可能还是会导致 volatile 变量可见性问题，所以老的旧内存模型没有从根本上解决 volatile 变量的可见性问题。在新的内存模型下，仍然是不允许对 volatile 变量进行 reorder 的，不同的是再也不轻易(虽然没有完全禁止掉)允许对它周围的非 volatile 变量进行排序。

由于上述第(4)条中对 volatile 字段以及周围非 volatile 字段(或变量)reorder 的限制，如下程序中，假设线程 A 正在执行 reader 方法，同时，线程 B 正在执行 writer 方法。线程 B 完成对 volatile 字段 v 的赋值后，相应的结果被写回内存。如果此时线程 A 便得到的 v 的值正好为 true，那么线程 A 也可以安全地引用 x 的值。然而，需要注意的是，假如 v 不是 volatile 的，那么上述结果就不一定了，因为 x 和 v 赋值的顺序可能被 reorder。

```
class VolatileSample1 {
    int x = 0;
    volatile boolean v = false;

    public void writer() {
        x = 42;
        v = true;
    }

    public void reader() {
        /* 由于volatile 的特点，这里要想 v 为 true，则 x 的肯定已经执行赋值(assign)动作
         * 且已写回(writer)主内存了，所以不会出现 v=true，x=0 的情形。但时，如果这里先
         * 访问的是 x 变量，则由于volatile 不具有原子性，则还是会出现 v=true，x=0 的情形，
         * 具体请看后面测试
         */
        if (v == true) {
            //uses x - 确保能看见 42.
        }
    }
}
```

假设一个线程调用 writer，一个线程调用 reader。写线程将 V 写回到主内存中，读线程从主内存中获取 v。因此，如果读的线程能够看到 v 的值为 true，这就能确保该读线程能看到 x 的值为 42，因为 x 是在 v 前面赋值，所以也会先写回到内存。如果 v 不是 volatile，编译器就可能在写回到主内存时对 v 与 x 进行 reader，这样的就可能在读的线程看到 v 为 true 时，x 却还是为 0，因为写线程对写回主内存动作进行重新 reader 过了。

而下面是对 volatile 变量的测试：

```
class VolatileSample2 {
    int x = 0;
    volatile boolean v = false;
    String result;

    public void writer() {
        x = 42;
        v = true;
    }
```

```
    public void reader() {
        /*
         * 如果是 result-"x-"+x+",v-"+v;，则快就会出现 x-0,v-true 这样的结果，因为这
         * 条语句完全可能先访问 x 后，另外线程再执行 writer 方法，待 writer 方法执行完成后，
         * 再接着访问 v，此时就会出现 x=0,v=true 的结果;
         *
         * 但如果是 result="v="+v+",x="+x;，则要想出现 v=true,x=0 的结果，则一定要等
         * writer 方法执行完并写回到主内存后再执行 reader 方法，由于 v 声明的是 volatile 变
         * 量，volatile 变量会禁止 reorder 任意两个者 volatile 变量，并且同时严格限制
         * reorder volatile 变量周围 的非 volatile 变量，所以由于 x 比 v 前赋值，则写回主
         * 存时也会一定按照此顺序，所以当 v 为 true 时，则主存中的 x 肯定是 42，绝不会是 0(
         * 这里不要注意的是，volatile 的变量也会在读取主内存时严格按照程序的顺序执行，
         * 所以这里根本不会先访问 x 再 v 的可能，如果这那样，则也会出现 v=true,x=0 的结果)。
         *
         * 这里只将 x 声明成 volatile 其结果也是一样的。当然如果两个都声明成 volatile 时，
         * 会更安全，因为这里会完全“禁止”重排，而一个的话只是“严格限制”而已，可能还是
         * 不会很安全，所以一般 两个都设置为最安全。
         *
         * 当 x,v 都是 volatile 时， result="x="+x+",v="+v;执行的结果还是有可能为
         * x=0,v=true， 因为 volatile 只是保证了可见性与顺序性两个特点为，但并不能保证
         * 原子性。此种情况下要得到 x=0,v=true，只需 reader 方法先执行，等访问 x 完后而 v
         * 还未访问时，开始调试 writer 方法， 待 writer 整个方法执行完后并将 x,v 写回主内存
         * 后，再执行 reader 方法，继续访问 v，此时的结 果就是 x=0,v=true。 另外，
         * result="x="+x+",v="+v;也会严格按照程序的顺序来执行访问操作(即 volatile 不
         * 只是在写回内存时是按程序语义的执行顺序来执行，在读的时候也是这样 要按照程序
         * 的访问顺序来，但如果不是 volatile 变量时，则 read 动作就可能不会按照程序顺序来
         * 执行，但这好像对纯粹的访问操作没有什么影响，这好只有访问操作的不变对象一样，
         *不会出现线程不安全的问题)，即先访问 x 后再能访问 v，但这绝不是原子性的，很
         * 有可能从他们中 间 切换到其他线程。

         * 另外，在测试的过程中发现 writer 方法的原子性要比 reader 的原子性要强，即多个访
         * 问操作在 一起不如多个赋值语句原子性强
         */
        result = "x=" + x + ",v=" + v;
        //result = "v=" + v + ",x=" + x;
    }

}

public class VolatileTest {
    public static void main(String[] args) {
        while (!Thread.currentThread().isInterrupted()) {
            final VolatileSample2 s = new VolatileSample2();
            final Thread w = new Thread(){
                public void run() {
                    s.writer();
                }
            };

            final Thread r = new Thread(){
                public void run() {
                    s.reader();
                }
            };
```

```
                r.start();
                w.start();
                new Thread(){
                    public void run() {
                        try {
                            w.join();
                            r.join();
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                        if (s.result.equals("x=0,v=true")) {
                            System.out.println(this.getName() + " " +
s.result);
                            System.exit(0);
                        }
                    }
                }.start();

                Thread.yield();
            }
        }
}
```

双重检测在新的内存模型下能很好地工作吗？看下面的代码：

```
private static Something instance = null;

public Something getInstance() {
  if (instance == null) {
    synchronized (this) {
      if (instance == null)
        instance = new Something();//1
    }
  }
  return instance;
}
```

首先，如果上面程序不加任何修改，这个在旧的或是新的内存模型下都不能正确的工作。上面程序 //1 处在多线程的情况下会有问题，如果一个线程在 //1 处已调用完构造器，但 Something 的实例域可能还没有被写回到内存，而在这之前会将创建好的对象(但并非初始化完全的对象，因为没有将它的实例域完全写回到主内存)赋值给了 instance 引用，这样另一个线程拿到的 instance 所指向的对象其实是不完整的，即所指向的对象的实例域还不可见，这样在使用这个 instnace 时就会有问题。在 1.5 或之后的版本中，我们可以将 instance 设置为 volatile 就可以了，这样就会确保将实例域的数据写回到主内存的动作在将实例赋值给 instance 引用动作之前发生(即 volatile 的 happens-before 规则)，所以这样就确保了在使用前对象已完全初始化完成。

而在新的内存模型下怎么才能使用 final 域正常的工作呢？JSR 133 新的目标中提出了一个初始化安全的新保障：如果一个对象被安全、适当地构造(在构造器中将当前正在构造的对象 this 暴露给外界是不安全的，“安全构造”技术请参考这里)，这样其他线程可以在不使用同步的情况下看到该对象在构造器里设置的 final 域的值。在构造期间，不要公布

“this”引用，即在构造函数完成之前，使 this 引用暴露给另一个线程，这种暴露可能是显式的，也可能是隐式的。

```
class FinalFieldExample {
  final int x;
  int y;
  static FinalFieldExample f;
  public FinalFieldExample() {
    x = 3;
    y = 4;
  }

  static void writer() {
    f = new FinalFieldExample();
  }

  static void reader() {
    if (f != null) {
      int i = f.x;
      int j = f.y;
    }
  }
}
```

上面的类中展示了怎么使用 final 域。能够确保另一调用 reader 的线程它能看到 f.x 的值是因为 f.x 是 final 的，但不能保证它能看到 f.y 的值是 4，因为它不是 final 的。如果 FinalFieldExample 的构造器像这样：

```
public FinalFieldExample() { // bad!
  x = 3;
  y = 4;
  // bad construction - allowing this to escape
  global.obj = this;// 暴露 this
}
```

这样其他线程通过 global.obj 读 x 的值将不能确保是 3。上面列举的例子是 final 类型的基本类型变量，如果 final 修饰的是一个引用类型，则也会有这样的保障：在拿到 final 引用类型前这个引用所指向的对象的所有域将完全初始化构造完成。

16.3.5　long 和 double 型变量

Java 内存模型要求 lock、unlock、read、load、assign、use、store 和 write 这 8 个操作都具有原子性，但是对于 64 位的数据类型(long 和 double)，在模型中特别定义了一条宽松的规定：允许虚拟机将没有被 volatile 修饰的 64 位数据的读写操作划分为两次 32 位的操作来进行，即允许虚拟机实现选择可以不保证 64 位数据类型的 load、store、read 和 write 这四个操作的原子性，这点就是所谓的 long 和 double 的非原子性协定(Nonatomic Treatment of double and long Variables)。如果有多个线程共享一个并未声明为 volatile 的 long 或 double 类型的变量，并且同时对它们进行读取和修改操作，那么某些线程可能会读取到一个既非原值，也不是其他线程修改值的代表了“半个变量”的数值。

不过这种读取到“半个变量”的情况非常罕见，因为Java内存模型虽然允许虚拟机不把long和double变量的读写实现成原子操作，但允许虚拟机选择把这些操作实现为具有原子性的操作，而且还“强烈建议”虚拟机这样实现。在实际开发中，目前各种平台下的商用虚拟机几乎都选择把64位数据的读写操作作为原子操作来对待，因此我们在编写代码时一般不需要将用到的long和double变量专门声明为volatile。

16.3.6 原子性、可见性与有序性

介绍完Java内存模型的相关操作和规则，我们再整体回顾一下这个模型的特征。Java内存模型是围绕着在并发过程中如何处理原子性、可见性和有序性这三个特征来建立的，我们逐个来看一下哪些操作实现了这三个特性。

1) 原子性(Atomicity)

由Java内存模型来直接保证的原子性变量操作包括read、load、assign、use、store和write这六个，我们大致可以认为基本数据类型的访问读写是具备原子性的(10ng和double的非原子性协定例外，笔者的观点是知道这件事情就可以了，无须太过在意这些几乎不会发生的例外情况)。

如果应用场景需要一个更大范围的原子性保证(经常会遇到)，Java内存模型还提供了lock和unlock操作来满足这种需求，尽管虚拟机未把lock和unlock操作直接开放给用户使用，但是却提供了更高层次的字节码指令monitorenter和monitorexit来隐式地使用这两个操作，这两个字节码指令反映到Java代码中就是同步块-synchronized关键字，因此在synchronized块之间的操作也具备原子性。

2) 可见性(Visibility)

可见性就是指当一个线程修改了共享变量的值，其他线程能够立即得知这个修改。上文在讲解volatile变量的时候我们已详细讨论过这一点。Java内存模型是通过在变量修改后将新值同步回主内存，在变量读取前从主内存刷新变量值这种依赖主内存作为传递媒介的方式来实现可见性的，无论是普通变量还是volatile变量都是如此，普通变量与volatile变量的区别是volatile的特殊规则保证了新值能立即同步到主内存，以及每次使用前立即从主内存刷新。因此我们可以说volatile保证了多线程操作时变量的可见性，而普通变量则不能保证这一点。

除了volatile之外，Java还有两个关键字能实现可见性，它们是synchronized和final。同步块的可见性是由“对 -个变量执行unlock操作之前，必须先把此变量同步回主内存中(执行store和write操作)”这条规则获得的，而final关键字的可见性是指：被final修饰的字段在构造器中 -旦被初始化完成，并且构造器没有把“this”的引用传递出去(this引用逃逸是一件很危险的事情，其他线程有可能通过这个引用访问到“初始化了一半”的对象)，那么在其他线程中就能看见final字段的值。

3) 有序性(Ordering)

Java内存模型的有序性在前面讲解volatile时也详细地讨论过了，Java程序中天然的有序性可以总结为一句话：如果在本线程内观察，所有的操作都是有序的；如果在一个线程中观察另一个线程，所有的操作都是无序的。前半句是指“线程内表现为串行的语义”

(Within-Thread As-If-Serial Semantics)，后半句是指“指令重排序”现象和“工作内存与主内存同步延迟”现象。

Java 语言提供了 volatile 和 synchronized 两个关键字来保证线程之间操作的有序性，volatile 关键字本身就包含了禁止指令重排序的语义，而 synchronized 则是由“一个变量在同一个时刻只允许一条线程对其进行 lock 操作”这条规则获得的，这个规则决定了持有同一个锁的两个同步块只能串行地进入。

介绍完并发的三种重要特性，读者有没有发现 synchronized 关键字在需要这三种特性的时候都可以作为其中一种的解决方案？看起来很“万能”吧？的确，大部分的并发控制操作都能使用 synchronized 来完成。synchronized 的“万能”也间接造就了它被程序员滥用的局面，越“万能”的并发控制，通常会伴随着越大的性能影响

16.3.7　先行发生原则

如果 Java 内存模型中所有的有序性都只靠 volatile 和 synchronized 来完成，那么有一些操作将会变得很啰嗦，但是我们在编写 Java 并发代码的时候并没有感觉到这一点，这是因为 Java 语言中有一个“先行发生”(Happens-before)的原则。这个原则非常重要，它是判断数据是否存在竞争，线程是否安全的主要依据，依赖这个原则，我们可以通过几条规则一揽子解决并发环境下两个操作之间是否可能存在冲突的所有问题。

现在就来看看“先行发生”原则指的是什么。先行发生是 Java 内存模型中定义的两项操作之间的偏序关系，如果说操作 A 先行发生于操作 B，其实就是说在发生操作 B 之前，操作 A 产生的影响能被操作 B 观察到，“影响”包括修改了内存中共享变量的值、发送了消息、调用了方法等。这句话不难理解，但它意味着什么呢？我们可以举个例子来说明一下，例如下面列出了这三句伪代码，演示了先行发生原则的过程。

```
//以下操作在线程 A 中执行
i=1;
{
//以下操作在线程 B 中执行
j=i;
//以下操作在线程 c 中执行      ?
i= 2;
```

假设线程 A 中的操作“i-l”先行发生于线程 B 的操作“j=1”那我们就可以确定在线程 B 的操作执行后，变量 j 的值 -定是等于 1，得出这个结论的依据有两个，-是根据先行发生原则，“i 1”的结果可以被观察到；二是线程 C 登场之前，线程 A 操作结束之后没有其他线程会修改变量 i 的值。现在再来考虑线程 C，我们依然保持线程 A 和 B 之间的先行发生关系，而 C 出现在线程 A 和 B 的操作之间，但是 C 与 B 没有先行发生关系，那 j 的值会是多少呢？答案是不确定 1、1 和 2 都有可能，因为线程 C 对变量 i 的影响可能会被线程 B 观察到，也可能不会，这时候线程 B 就存在读取到过期数据的风险，不具备多线程安全性。

下面是 Java 内存模型下一些“天然的”先行发生关系，这些先行发生关系无须任何同步器协助就已经存在，可以在编码中直接使用。如果两个操作之间的关系不在此列，并且

无法从下列规则推导出来的话，它们就没有顺序性保障，虚拟机可以对它们进行随意地重排序。

- 程序次序规则(Program Order Rule)：在一个线程内，按照程序代码顺序，书写在前面的操作先行发生于书写在后面的操作。准确地说应该是控制流顺序而不是程序代码顺序，因为要考虑分支、循环等结构。
- 管程锁定规则(Monitor Lock Rule)：一个 unlock 操作先行发生于后面对同一个锁的 lock 操作。这里必须强调的是同一个锁，而“后面”是指时间上的先后顺序。
- volatile 变量规则(Volatile Variable Rule)：对一个 volatile 变量的写操作先行发生于后面对这个变量的读操作，这里的“后面”同样是指时间上的先后顺序。
- 线程启动规则(Thread Start Rule)：Thread 对象的 start()方法先行发生于此线程的每一个动作。
- 线程终止规则(Thread Termination Rule)：线程中的所有操作都先行发生于对此线程的终止检测，我们可以通过 Thr<:ad.join()方法结束、Thread.isAlive()的返回值等手段检测到线程已经终止执行。
- 线程中断规则(Thread Interruption Rule)：对线程 interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生，可以通过 Thread.interrupted()方法检测到是否有中断发生。
- 对象终结规则(Finalizer Rule)：一个对象的初始化完成(构造函数执行结束)先行发生于它的 finalize()方法的开始。
- 传递性(Transitivity)：如果操作 A 先行发生于操作 B，操作 B 先行发生于操作 C，那就可以得出操作 A 先行发生于操作 C 的结论。

Java 语言无须任何同步手段保障就能成立的先行发生规则就只有上面这些了，笔者演示一下如何使用这些规则去判定操作间是否具备顺序性，对于读写共享变量的操作来说，就是线程是否安全，读者还可以从下面这个例子中感受一下“时间上的先后顺序”与“先行发生”之间有什么不同。

16.4 线　　程

并发不一定要依赖多线程(如 PHP 中很常见的多进程并发)，但是在 Java 里面谈论并发，大多数都与线程脱不开关系。既然我们这本书探讨的话题是 Java 虚拟机的特性，那讲到 Java 线程，我们就从 Java 线程在虚拟机中的实现开始讲起。

16.4.1 线程的实现

我们知道，线程是比进程更轻量级的调度执行单位，线程的引入，可以把一个进程的资源分配和执行调度分开，各个线程既可以共享进程资源(内存地址、文件 I/O 等)，又可以独立调度(线程是 CPU 调度的最基本单位)。主流的操作系统都提供了线程实现，Java 语言则提供了在不同硬件和操作系统平台下对线程操作的统一处理，每个 java.lang.Thread 类

的实例就代表了一个线程。不过 Thread 类与大部分的 Java API 有着显著的差别，它的所有关键方法都被声明为 Native。

在 Java API 中一个 Native 方法可能就意味着这个方法没有使用或无法使用平台无关的手段来实现(当然也可能是为了执行效率而使用 Native 方法，不过通常最高效率的手段也就是平台相关的手段)。正因为这个原因，作者把本节的标题定为“线程的实现”而不是“Java 线程的实现”。

实现线程主要有三种方式具体说明如下。

1．使用内核线程实现

内核线程(Kernel Thread，KLT)就是直接由操作系统内核(Kernel，下称内核)支持的线程，这种线程由内核来完成线程切换，内核通过操纵调度器(Scheduler)对线程进行调度，并负责将线程的任务映射到各个处理器上。每个内核线程都可以看做是内核的一个分身，这样操作系统就有能力同时处理多件事情，支持多线程的内核就叫多线程内核(Multi-Threads Kernel)。

程序一般不会直接去使用内核线程，而是去使用内核线程的一种高级接口——轻量级进程(Light Weight Process，LWP)，轻量级进程就是我们通常意义上所讲的线程，由于每个轻量级进程都由一个内核线程支持，因此只有先支持内核线程，才能有轻量级进程。这种轻量级进程与内核线程之间 1∶1 的关系称为一对一的线程模型。由于内核线程的支持，每个轻量级进程都成为一个独立的调度单元，即使有一个轻量级进程在系统调用中阻塞了，也不会影响整个进程继续工作，但是轻量级进程具有它的局限性：首先，由于是基于内核线程实现的，所以各种线程操作，如创建、析构及同步，都需要进行系统调用。而系统调用的代价相对较高，需要在用户态(User Mode)和内核态(Kernel Mode)中来回切换。其次，每个轻量级进程都需要有一个内核线程的支持，因此轻量级进程要消耗一定的内核资源(如内核线程的栈空间)，因此一个系统支持轻量级进程的数量是有限的。

2．使用用户线程实现

广义上来讲，一个线程只要不是内核线程，那就可以认为是用户线程(User Thread，UT)，因此从这个定义上来讲轻量级进程也属于用户线程，但轻量级进程的实现始终是建立在内核之上的，许多操作都要进行系统调用，因此效率会受到限制。

而狭义上的用户线程指的是完全建立在用户空间的线程库上，系统内核不能感知到线程存在的实现。用户线程的建立、同步、销毁和调度完全在用户态中完成，不需要内核的帮助。如果程序实现得当，这种线程不需要切换到内核态，因此操作可以是非常快速且低消耗的，也可以支持规模更大的线程数量，部分高性能数据库中的多线程就是由用户线程实现的。这种进程与用户线程之间 1∶N 的关系称为一对多的线程模型。

使用用户线程的优势在于不需要系统内核支援，劣势也在于没有系统内核的支援，所有的线程操作都需要用户程序自己处理。线程的创建、切换和调度都是需要考虑问题，而且由于操作系统只把处理器资源分配到进程，那诸如“阻塞如何处理”、“多处理器系统中如何将线程映射到其他处理器上”这类问题解决起来将会异常困难，甚至不可能完成。因而使用用户线程实现的程序一般都比较复杂，除了以前在不支持多线程的操作系统中(如

DOS)的多线程程序与少数有特殊需求的程序外，现在使用用户线程的程序越来越少了，Java、Ruby 等语言都曾经使用过用户线程，最终又都放弃了使用它。

3．混合实现

线程除了依赖内核线程实现和完全由用户程序自己实现之外，还有一种将内核线程与用户线程一起使用的实现方式。在这种混合实现下，既存在用户线程，也存在轻量级进程。用户线程还是完全建立在用户空间中，因此用户线程的创建、切换、析构等操作依然廉价，并且可以支持大规模的用户线程并发。而操作系统提供支持的轻量级进程则作为用户线程和内核线程之间的桥梁，这样可以使用内核提供的线程调度功能及处理器映射，并且用户线程的系统调用要通过轻量级线程来完成，大大降低了进程被阻塞的风险。在这种混合模式中，用户线程与轻量级进程的数量比是不定的，是 M：N 的关系，这种就是多对多的线程模型。许多 Unix 系列的操作系统，如 Solaris、HP-UX 等都提供了 M：N 的线程模型实现。

4．Java 线程的实现

Java 线程在 JDK 1.2 之前，是基于名为“绿色线程”(Green Threads)的用户线程实现的，而在 JDK 1.2 中，线程模型被替换为基于操作系统原生线程模型来实现。因此在目前的 JDK 版本中，操作系统支持怎样的线程模型，在很大程度上就决定了 Java 虚拟机的线程是怎样映射的，这点在不同的平台上没有办法达成一致，虚拟机规范中也并未限定 Java 线程需要使用哪种线程模型来实现。线程模型只对线程的并发规模和操作成本产生影响，对 Java 程序的编码和运行过程来说，这些差异都是透明的。

对于 Sun JDK 来说，它的 Windows 版与 Linux 版都是使用一对一的线程模型来实现的，一条 Java 线程就映射到一条轻量级进程之中，因为 Windows 和 Linux 系统提供的线程模型就是一对一的。

而在 Solaris 平台中，由于操作系统的线程特性可以同时支持一对一(通过 Bound Threads 或 Alternate Libthread 实现)及多对多(通过 LWP/Thread Based Synchronization 实现)的线程模型，因此在 Solaris 版的 JDK 中也对应提供了两个平台专有的虚拟机参数：-XX:+UseLWPSynchronization(默认值)和-XX:+UseBoundThreads 来明确指定虚拟机使用的是哪种线程模型。

16.4.2 线程调度

计算机通常只有一个 CPU，在任意时刻只能执行一条机器指令，每个线程只有获得 CPU 的使用权才能执行指令。

所谓多线程的并发运行，其实是指各个线程轮流获得 CPU 的使用权，分别执行各自的任务。在可运行池中，会有多个处于就绪状态的线程在等待 CPU。

Java 虚拟机的一项任务就是负责线程的调度。线程的调度是指按照特定的机制为多个线程分配 CPU 的使用权，有两种调度模型：分时调度模型和抢占式调度模型。

分时调度模型是指让所有线程轮流获得 CPU 的使用权，并且平均分配每个线程占用 CPU 的时间片。Java 虚拟机采用抢占式调度模型，它是指优先让可运行池中优先级高的线

程占用 CPU，如果可运行池中线程的优先级相同，那么就随机地选择一个线程，使其占用 CPU。处于运行状态的线程会一直运行，直至它不得不放弃 CPU。一个线程会因为以下原因而放弃 CPU：

- Java 虚拟机让当前线程暂时放弃 CPU，转到就绪状态，使其他线程获得运行机会。
- 当前线程因为某些原因而进入阻塞状态。
- 线程运行结束。

值得注意的是，线程的调度不是跨平台的，它不仅取决于 Java 虚拟机，还依赖于操作系统。在某些操作系统中，只要运行中的线程没有遇到阻塞，就不会放弃 CPU；在某些操作系统中，即使运行中的线程没有遇到阻塞，也会在运行一段时间后放弃 CPU，给其他线程运行的机会。

由于 Java 线程的调度不是分时的，因此同时启动多个线程后，不能保证各个线程轮流获得均等的 CPU 时间片。如果程序希望干预 Java 虚拟机对线程的调度过程，从而明确地让一个线程给另外一个线程运行的机会，可以采取以下办法之一：

(1) 调整各个线程的优先级。

(2) 让处于运行状态的线程调用 Thread.sleep()方法。

(3) 让处于运行状态的线程调用 Thread.yield()方法。

(4) 让处于运行状态的线程调用另一个线程的 join()方法。

16.4.3　线程状态间的转换

线程的状态转换是线程控制的基础。线程状态总的可分为五大状态：分别是生、死、可运行、运行、等待/阻塞。具体过程如图 16-2 所示。

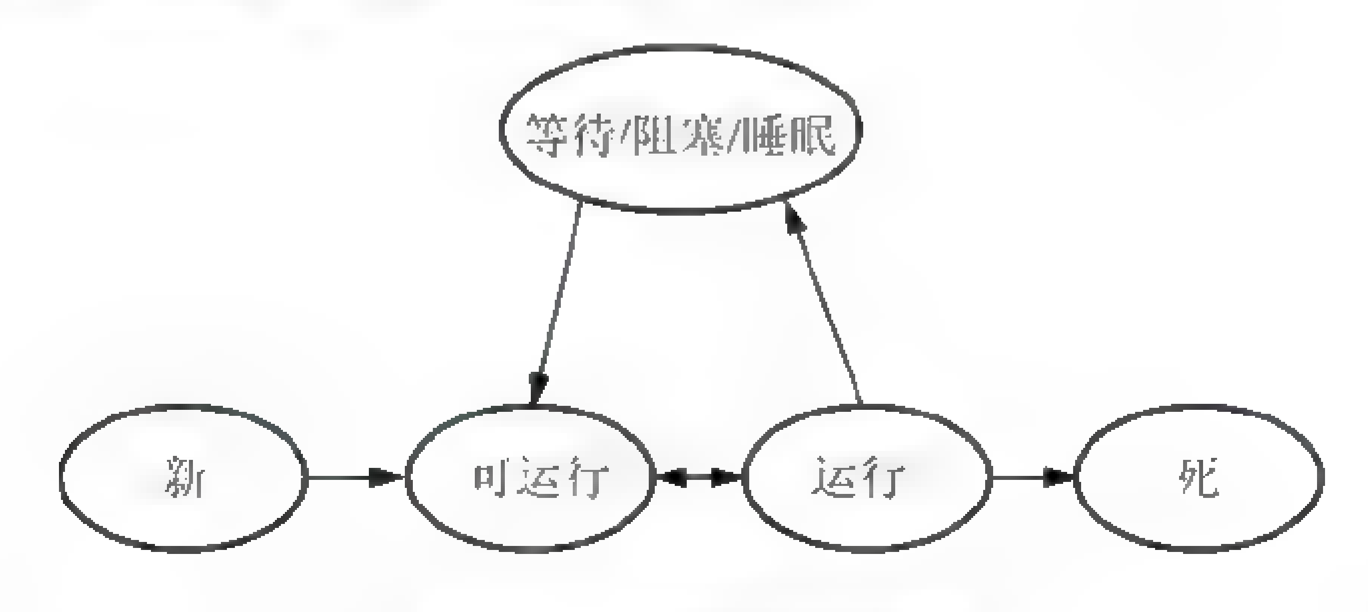


图 16-2　线程的状态转换

(1) 新状态：线程对象已经创建，还没有在其上调用 start()方法。

(2) 可运行状态：当线程有资格运行，但调度程序还没有把它选定为运行线程时线程所处的状态。当 start()方法调用时，线程首先进入可运行状态。在线程运行之后或者从阻塞、等待或睡眠状态回来后，也返回到可运行状态。

(3) 运行状态：线程调度程序从可运行池中选择一个线程作为当前线程时线程所处的状态。这也是线程进入运行状态的唯一一种方式。

(4) 等待/阻塞/睡眠状态：这是线程有资格运行时它所处的状态。实际上这个三状态组合为一种，其共同点是：线程仍旧是活的，但是当前没有条件运行。换句话说，它是可运

行的，但是如果某件事件出现，他可能返回到可运行状态。

(5) 死亡态：当线程的 run()方法完成时就认为它死去。这个线程对象也许是活的，但是，它已经不是一个单独执行的线程。线程一旦死亡，就不能复生。 如果在一个死去的线程上调用 start()方法，会抛出 java.lang.IllegalThreadStateException 异常。

对于线程阻止需要考虑以下三个方面，不考虑 IO 阻塞的情况：

- 睡眠；
- 等待；
- 因为需要一个对象的锁定而被阻塞。

1) 睡眠

Thread.sleep(long millis)和 Thread.sleep(long millis, int nanos)静态方法强制当前正在执行的线程休眠(暂停执行)，以“减慢线程”。当线程睡眠时，它入睡在某个地方，在苏醒之前不会返回到可运行状态。当睡眠时间到期，则返回到可运行状态。

(1) 线程睡眠的原因

线程执行太快，或者需要强制进入下一轮，因为 Java 规范不保证合理的轮换。例如下面是睡眠的实现的代码，在此调用了静态方法。

```
try {
   Thread.sleep(123);
} catch (InterruptedException e) {
   e.printStackTrace();
}
```

(2) 睡眠的位置

为了让其他线程有机会执行，可以将 Thread.sleep()的调用放线程 run()之内。这样才能保证该线程执行过程中会睡眠。例如，在前面的例子中，将一个耗时的操作改为睡眠，以减慢线程的执行。可以这么写：

```
public void run() {
   for(int i = 0;i<5;i++){
// 很耗时的操作，用来减慢线程的执行
//         for(long k= 0; k <100000000;k++);
       try {
          Thread.sleep(3);
       } catch (InterruptedException e) {
          e.printStackTrace();  .
       }
       System.out.println(this.getName()+" :"+i);
   }
}
```

运行结果：

```
阿三 :0
李四 :0
阿三 :1
阿三 :2
阿三 :3
李四 :1
```

```
李四 :2
阿三 :4
李四 :3
李四 :4
Process finished with exit code 0
```

这样，线程在每次执行过程中，总会睡眠 3 毫秒，睡眠了，其他的线程就有机会执行了。

注意：

- 线程睡眠是帮助所有线程获得运行机会的最好方法。
- 线程睡眠到期自动苏醒，并返回到可运行状态，不是运行状态。sleep()中指定的时间是线程不会运行的最短时间。因此，sleep()方法不能保证该线程睡眠到期后就开始执行。
- sleep()是静态方法，只能控制当前正在运行的线程。

下面给个例子：

```
/**
* 一个计数器，计数到 100，在每个数字之间暂停 1 秒，每隔 10 个数字输出一个字符串
*
* @author leizhimin 2008-9-14 9:53:49
*/
public class MyThread extends Thread {

    public void run() {
        for (int i = 0; i < 100; i++) {
            if ((i) % 10 == 0) {
                System.out.println("-------" + i);
            }
            System.out.print(i);
            try {
                Thread.sleep(1);
                System.out.print("    线程睡眠 1 毫秒! \n");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) {
        new MyThread().start();
    }
}
```

执行后输出：

```
-------0
0    线程睡眠 1 毫秒!
1    线程睡眠 1 毫秒!
2    线程睡眠 1 毫秒!
3    线程睡眠 1 毫秒!
4    线程睡眠 1 毫秒!
5    线程睡眠 1 毫秒!
```

```
6    线程睡眠 1 毫秒！
7    线程睡眠 1 毫秒！
8    线程睡眠 1 毫秒！
9    线程睡眠 1 毫秒！
         10
……
         90
90    线程睡眠 1 毫秒！
91    线程睡眠 1 毫秒！
92    线程睡眠 1 毫秒！
93    线程睡眠 1 毫秒！
94    线程睡眠 1 毫秒！
95    线程睡眠 1 毫秒！
96    线程睡眠 1 毫秒！
97    线程睡眠 1 毫秒！
98    线程睡眠 1 毫秒！
99    线程睡眠 1 毫秒！

Process finished with exit code 0
```

2) 线程的优先级和线程让步 yield()

线程的让步是通过 Thread.yield()来实现的。yield()方法的作用是：暂停当前正在执行的线程对象，并执行其他线程。要理解 yield()，必须了解线程的优先级的概念。线程总是存在优先级，优先级范围在 1~10 之间。JVM 线程调度程序是基于优先级的抢先调度机制。在大多数情况下，当前运行的线程优先级将大于或等于线程池中任何线程的优先级。但这仅仅是大多数情况。

当设计多线程应用程序的时候，一定不要依赖于线程的优先级。因为线程调度优先级操作是没有保障的，只能把线程优先级作用作为一种提高程序效率的方法，但是要保证程序不依赖这种操作。

当线程池中线程都具有相同的优先级，调度程序的 JVM 实现自由选择它喜欢的线程。这时候调度程序的操作有两种可能：一是选择一个线程运行，直到它阻塞或者运行完成为止。二是时间分片，为池内的每个线程提供均等的运行机会。

设置线程的优先级：线程默认的优先级是创建它的执行线程的优先级。可以通过 setPriority(int newPriority)更改线程的优先级。例如：

```
Thread t = new MyThread();
t.setPriority(8);
t.start();
```

线程优先级为 1~10 之间的正整数，JVM 从不会改变一个线程的优先级。然而，1~10 之间的值是没有保证的。一些 JVM 可能不能识别 10 个不同的值，而将这些优先级进行每两个或多个合并，变成少于 10 个的优先级，则两个或多个优先级的线程可能被映射为一个优先级。

线程默认优先级是 5，Thread 类中有三个常量，定义线程优先级范围：

- static int MAX_PRIORITY：线程可以具有的最高优先级。
- static int MIN_PRIORITY：线程可以具有的最低优先级。
- static int NORM_PRIORITY：分配给线程的默认优先级。

3) Thread.yield()方法

Thread.yield()方法作用是：暂停当前正在执行的线程对象，并执行其他线程。yield()应该做的是让当前运行线程回到可运行状态，以允许具有相同优先级的其他线程获得运行机会。因此，使用 yield()的目的是让相同优先级的线程之间能适当的轮转执行。但是，实际中无法保证 yield()达到让步目的，因为让步的线程还有可能被线程调度程序再次选中。

由此可见，yield()从未导致线程转到等待/睡眠/阻塞状态。在大多数情况下，yield()将导致线程从运行状态转到可运行状态，但有可能没有效果。

4) join()方法

Thread 的非静态方法 join()让一个线程 B“加入”到另外一个线程 A 的尾部。在 A 执行完毕之前，B 不能工作。例如：

```
Thread t = new MyThread();
t.start();
t.join();
```

另外，join()方法还有带超时限制的重载版本。例如 t.join(5000);则让线程等待 5000 毫秒，如果超过这个时间，则停止等待，变为可运行状态。线程的加入 join()对线程栈导致的结果是线程栈发生了变化，当然这些变化都是瞬时的。

Java

第 17 章

安全和优化合二为一

自从计算机诞生之日起，安全问题就一直是人们关心的话题。在大型商业应用领域中，安全和优化一直是研究并发展的重点。本章将详细讲解 JVM 安全和优化方面的基本问题。

17.1 线程安全

线程安全指的是当多个线程操作同一个数据段时，用相应的互斥机制，避免数据段中的数据错误。

每个线程尽量只访问别的线程不访问的变量或内存，如果硬是要访问同一变量或内存的话，就要采用适当的互斥机制来避免由于线程切换而导致的不确定性。“线程安全函数”就是当你在多线程程序中调用该函数，该函数本身不会出错，并且能得到正确的结果或处理。

17.1.1 Java 中的线程安全

我们已经有了线程安全的一个抽象定义，那接下来我们就讨论一下在 Java 语言中，线程安全具体是如何体现的？有哪些操作是线程安全的？我们这里讨论的线程安全，就限定于多个线程之间存在共享数据访问这个前提，因为如果一段代码根本不会与其他线程共享数据，那么从线程安全的角度上看，程序是串行执行还是多线程执行对它来说是完全没有区别的。

为了更深入地理解线程安全，在这里我们可以不把线程安全当作一个非真即假的二元排它选项来看待，按照线程安全的“安全程度”由强至弱来排序，我们可以将 Java 语言中各种操作共享的数据分为以下五类：不可变、绝对线程安全、相对线程安全、线程兼容和线程对立。

1. 不可变

在 Java 语言里面(特指 JDK 1.5 以后，即 Java 内存模型被修正之后的 Java 语言)，不可变(Immutable)的对象一定是线程安全的，无论是对象的方法实现还是方法的调用者，都不需要再进行任何的线程安全保障措施，在上一章里我们谈到过 final 关键字带来的可见性时曾经提到过这一点，只要一个不可变的对象被正确地构建出来(没有发生 this 引用逃逸的情况)，那其外部的可见状态永远也不会改变，永远也不会看到它在多个线程之中处于不一致的状态。“不可变”带来的安全性是最简单最纯粹的。

Java 语言中，如果共享数据是一个基本数据类型，那么只要在定义时使用 final 关键字修饰它就可以保证它是不可变的。如果共享数据是一个对象，那就需要保证对象的行为不会对其状态产生任何影响才行，如果读者还没想明白这句话，不妨想一想 java.lang.String 类的对象，它是一个典型的不可变对象，我们调用它的 substring()、replace()和 concat()这些方法都不会影响它原来的值，只会返回一个新构造的字符串对象。

保证对象行为不影响自己状态的途径有很多种，其中最简单的就是把对象中带有状态的变量都声明为 final，这样在构造函数结束之后，它就是不可变的，例如演示代码 17-1 中 java.lang.Integer 构造函数，它通过将内部状态变量 value 定义为 final 来保障状态不变。

演示代码 17-1　JDK 中 Integer 类的构造函数

```
private final int value;
public Integer(int value) {
this.value=value;
}
```

在 Java API 中符合不可变要求的类型，除了上面提到的 String 之外，常用的还有枚举类型，以及 java.lang.Number 的部分子类，如 Long 和 Double 等数值包装类型，BigInteger 和 BigDecimal 等大数据类型；但同为 Number 的子类型的原子类 AtomicInteger 和 AtomicLong 则并非不可变的，读者不妨看看这两个原子类的源码，想一想为什么。

2. 绝对线程安全

绝对的线程安全完全满足 Brian Goetz 给出的线程安全的定义，这个定义其实是很严格的，一个类要达到"不管运行时环境如何，调用者都不需要任何额外的同步措施"通常需要付出很大的，甚至是不切实际的代价。在 Java API 中标注自己是线程安全的类，大多数都不是绝对的线程安全。我们可以通过 Java API 中一个不是"绝对线程安全"的线程安全类来看看这里的"绝对"是什么意思。

如果说 java.util.Vector 是一个线程安全的容器，相信所有的 Java 程序员对此都不会有异议，因为它的 add()、get()和 size()这类方法都是被 synchronized 修饰的，尽管这样效率很低，但确实是安全的。但是，即使它所有的方法都被修饰成同步，也不意味着调用它的时候永远都不再需要同步手段了，请看演示代码 17-2 中的测试代码。

演示代码 17-2　测试 Vector 线程安全

```
private static Vector<Integer> vector=new Vector<Integer>()j
public static void main(String[] args) {
while (true) f
for (int i=o; i<10; i++) {
vector.add (i);
}
Thread removeThread=new Thread(new Runnable() {
@Override
public void run() {
for (inti=o;i(vector.size(); i++) {
vector.remove (i);
}
}
});
Thread printThread=new Thread(new Runnable() {
@Override
public void run() {
for (int i=o; i(vector.size(); i++) {
system.out .println( (vector.get (i)));
}
}
});
removeThread.start();
printThread. start();
//不要同时产生过多的线程，否则会导致操作系统假死
```

```
while (Thread.activeCount() >20);
}
}
```

运行结果如下：

```
Exception in thread "Thread-172"
j ava .lang .ArrayIndexOutOfBoundsException:
Array index out of range: 17
at java .util.Vector.remove (Vector.java: 777)
at  org. fenixsoft .mulithread. VectorTest$1. run( VectorTest.j ava: 21)
at java .lang. Thread. run (Thread.java: 662)
```

很明显，尽管这里使用到的 Vector 的 get(). remove()和 size()方法都是同步的，但是在多线程的环境中，如果不在方法调用端做额外的同步措施，使用这段代码仍然是不安全的，因为如果另一个线程恰好在错误的时间里删除了一个元素，导致序号 i 已经不再可用的话，get()方法就会抛出一个 ArrayIndexOutOfBoundsException。如果要保证这段代码能正确地执行下去，我们不得不把 removeThread 和 printThread 的定义改成如演示代码 17-3 所示的这样。

演示代码 17-3　必须加入同步以保证 Vector 访问的线程安全性

```
Thread removeThread=new Thread(new Runnable() {
@Override
public void run() {
synchronized (vector) {
for (inti=o; i'vector.size(); i++) {
vector.remove (i);
}
}
}
};
Thread printThread=new Thread(new Runnable() {
@Override
public void run() {
synchronized (vector) {
for (int i=oj i<vector.size(); i++) {
System.out .println( (vector.get (i)));
}
}
}
};
```

3. 相对线程安全

相对的线程安全就是我们通常意义上所讲的线程安全，它需要保证对这个对象单独的操作是线程安全的，我们在调用的时候不需要做额外的保障措施，但是对于一些特定顺序的连续调用，就可能需要在调用端使用额外的同步手段来保证调用的正确性。演示代码 17-2 和演示代码 17-3 就是相对线程安全的一个很明显的案例。

在 Java 语言中，大部分的线程安全类都属于这种类型，例如 Vector、HashTable、Collections 的 synchronizedCollection()方法包装的集合等。



4. 线程兼容

线程兼容是指对象本身并不是线程安全的，但是可以通过在调用端正确地使用同步手段来保证对象在并发环境中安全地使用，我们平常说一个类不是线程安全的，绝大多数指的都是这种情况。Java API 中大部分的类都是线程兼容的，如与前面的 Vector 和 HashTable 相对应的集合类 ArrayList 和 HashMap 等。

5. 线程对立

线程对立是指不管调用端是否采取了同步措施，都无法在多线程环境中并发使用的代码。由于 Java 语言天生就具备多线程特性，线程对立这种排斥多线程的代码是很少出现的，而且通常都是有害的，应当尽量避免。

一个线程对立的例子是 Thread 类的 suspend()和 resume()方法，如果有两个线程同时持有一个线程对象，一个尝试去中断线程，一个尝试去恢复线程，如果并发进行的话，无论调用时是否进行了同步，目标线程都是存在死锁风险的，如果 suspend()中断的线程就是即将要执行 resume()的那个线程，那就肯定要产生死锁了。也正是由于这个原因，suspend()和 resume()方法已经被 JDK 声明废弃(@Deprecated)了。常见的线程对立的操作还有 System.setIn()、Sytem.setOut()和 System. runFinalizersOnExit()等。

17.1.2 线程安全的实现方法

了解了什么是线程安全之后，紧接着的一个问题就是我们应该如何实现线程安全，这听起来似乎是一件由代码如何编写来决定的事情，确实，如何实现线程安全与代码的编写有很大的关系，但虚拟机提供的同步和锁机制也起到了非常重要的作用。本节将介绍代码编写如何实现线程安全和虚拟机如何实现同步与锁，相对而言更偏重后者一些，只要读者了解了虚拟机线程安全手段的运作过程，自己去思考代码如何编写并不是一件困难的事情。

1. 互斥同步

互斥同步(Mutual Exclusion&Synchronization)是最常见的一种并发正确性保障手段，同步是指在多个线程并发访问共享数据时，保证共享数据在同一个时刻只被一条(或者是一些，使用信号量的时候)线程使用。而互斥是实现同步的一种手段，临界区(Critical Section)、互斥量(Mutex)和信号量(Semaphore)都是主要的互斥实现方式。

因此在这四个字里面，互斥是因，同步是果，互斥是方法，同步是目的。在 Java 里面，最基本的互斥同步手段就是 synchronized 关键字，synchronized 关键字经过编译之后，会在同步块的前后分别形成 monitorenter 和 monitorexit 这两个字节码指令，这两个字节码都需要一个 reference 类型的参数来指明要锁定和解锁的对象。如果 Java 程序中的 synchronized 明确指定了对象参数，那就是这个对象的 reference；如果没有明确指定，那就根据 synchronized 修饰的是实例方法还是类方法，去取对应的对象实例或 Class 对象来作为锁对象。

根据虚拟机规范的要求，在执行 monitorenter 指令时，首先要去尝试获取对象的锁。

如果这个对象没被锁定，或者当前线程已经拥有了那个对象的锁，把锁的计数器加 1，相应地，在执行 monitorexit 指令时会将锁计数器减 1，当计数器为 0 时，锁就被释放了。如果获取对象锁失败了，那当前线程就要阻塞等待，直到对象锁被另外一个线程释放为止。

在虚拟机规范对 monitorenter 和 monitorexit 的行为描述中，有两点是需要特别注意的。首先，synchronized 同步块对同一条线程来说是可重入的，不会出现自己把自己锁死的问题。其次，同步块在已进入的线程执行完之前，会阻塞后面其他线程的进入。上一章讲过，Java 的线程是映射到操作系统的原生线程之上的，如果要阻塞或唤醒一条线程，都需要操作系统来帮忙完成，这就需要从用户态转换到核心态中，因此状态转换需要耗费很多的处理器时间。对于代码简单的同步块(如被 synchronized 修饰的 getter()或 setter()方法)，状态转换消耗的时间可能比用户代码执行的时间还要长。所以 synchronized 是 Java 语言中一个重量级(Heavyweight)的操作，有经验的程序员都会在确实必要的情况下才使用这种操作。而虚拟机本身也会进行一些优化，譬如在通知操作系统阻塞线程之前加入一段自旋等待过程，避免频繁地切入到核心态之中。

除了 synchronized 之外，我们还可以使用 java.util.concurrent(下文称 J.U.C)包中的重入锁(ReentrantLock)来实现同步，在基本用法上，ReentrantLock 与 synchronized 很相似，他们都具备一样的线程重入特性，只是代码写法上有点区别，一个表现为 API 层面的互斥锁(lock()和 unlock()方法配合 try/finally 语句块来完成)，一个表现为原生语法层面的互斥锁。不过 ReentrantLock 比 synchronized 增加了一些高级功能，主要有三项：等待可中断、可实现公平锁，以及锁可以绑定多个条件。

- 等待可中断是指当持有锁的线程长期不释放锁的时候，正在等待的线程可以选择放弃等待，改为处理其他事情，可中断特性对处理执行时间非常长的同步块很有帮助。
- 可实现公平锁是指多个线程在等待同一个锁时，必须按照申请锁的时间顺序来依次获得锁；而非公平锁则不保证这一点，在锁被释放时，任何一个等待锁的线程都有机会获得锁。synchronized 中的锁是非公平的，ReentrantLock 默认情况下也是非公平的，但可以通过带布尔值的构造函数要求使用公平锁。
- 锁可以绑定多个条件是指一个 ReentrantLock 对象可以同时绑定多个 Condition 对象，而在 synchronized 中，锁对象的 wait()和 notify()或 notifyAll()方法可以实现一个隐含的条件，如果要和多于一个的条件关联的时候，就不得不额外地添加一个锁，而 ReentrantLock 则无需这样做，只需要多次调用 newCondition()方法即可。

如果需要使用到上述功能的时候，选用 ReentrantLock 是一个很好的选择，那如果是基于性能考虑呢？经过实践证明，多线程环境下 synchronized 的吞吐量下降得非常严重，而 ReentrantLock 则能基本保持在同一个比较稳定的水平上。与其说 ReentrantLock 性能好，倒还不如说 synchronized 还有非常大的优化余地。后续的技术发展也证明了这一点，JDK 1.6 中加入了很多针对锁的优化措施(下一节我们就会讲解这些优化措施)，JDK 1.6 发布之后，人们就发现 synchronized 与 ReentrantLock 的性能基本上是完全持平了。因此如果读者的程序是使用 JDK 1.6 部署的话，性能因素就不再是选择 ReentrantLock 的理由了，虚拟机在未来的性能改进中肯定也会更加偏向于原生的 synchronized，所以还是提倡在

synchronized 能实现需求的情况下，优先考虑使用 synchronized 来进行同步。

2. 非阻塞同步

互斥同步最主要的问题就是进行线程阻塞和唤醒所带来的性能问题，因此这种同步也被称为阻塞同步(Blocking Synchronization)。另外，它属于一种悲观的并发策略，总是认为只要不去做正确的同步措施(加锁)，那就肯定会出现问题，无论共享数据是否真的会出现竞争，它都要进行加锁(这里说的是概念模型，实际上虚拟机会优化掉很大一部分不必要的加锁)、用户态核心态转换、维护锁计数器和检查是否有被阻塞的线程需要被唤醒等操作。随着硬件指令集的发展，我们有了另外一个选择：基于冲突检测的乐观并发策略，通俗地说就是先进行操作，如果没有其他线程争用共享数据，那操作就成功了；如果共享数据有争用，产生了冲突，那就再进行其他的补偿措施(最常见的补偿措施就是不断地重试，直到试成功为止)，这种乐观的并发策略的许多实现都不需要把线程挂起，因此这种同步操作被称为非阻塞同步(Non-Blocking Synchronization)。

为什么笔者说使用乐观并发策略需要"硬件指令集的发展"才能进行呢？因为我们需要操作和冲突检测这两个步骤具备原子性，靠什么来保证呢？如果这里再使用互斥同步来保证就失去意义了，所以我们只能靠硬件来完成这件事情，硬件保证一个从语义上看起来需要多次操作的行为只通过一条处理器指令就能完成，这类指令常用的有：

- 测试并设置(Test-and-Set)；
- 获取并增加(Fetch-and-Increment)；
- 交换(Swap)；
- 比较并交换(Compare-and-Swap，下文称 CAS)；
- 加载链接/条件储存(Load-Linked / Store-Conditional，下文称 LL/SC)。

其中，前面的三条是上个世纪就已经存在于大多数指令集之中的处理器指令，后面的两条是现代处理器新增的，而且这两条指令的目的和功能是类似的。在 IA64、x86 指令集中通过 cmpxchg 指令完成 CAS 功能，在 sparc-TSO 中也有 casa 指令实现，而在 ARM 和 PowerPC 架构下，则需要使用一对 ldrex/strex 指令来完成 LL/SC 的功能。

CAS 指令需要有三个操作数，分别是内存位置(在 Java 中可以简单理解为变量的内存地址，用 V 表示)、旧的预期值(用 A 表示)和新值(用 B 表示)。CAS 指令执行时，当且仅当 V 符合旧预期值 A 时，处理器用新值 B 更新 V 的值，否则它就不执行更新，但是不管是否更新了 V 的值，都会返回 V 的旧值，上述的处理过程是一个原子操作。

在 JDK 1.5 之后，Java 程序中才可以使用 CAS 操作，该操作由 sun.misc.Unsafe 类里面的 compareAndSwapInt()和 compareAndSwapLong()等几个方法包装提供，虚拟机在内部对这些方法做了特殊处理，即时编译出来的结果就是一条平台相关的处理器 CAS 指令，没有方法调用的过程，或者可以认为是无条件内联进去了。

由于 Unsafe 类不是提供给用户程序调用的类(Unsafe.getUnsafe()的代码中限制了只有启动类加载器(Bootstrap ClassLoader)加载的 Class 才能访问它)，如果不采用反射手段，我们只能通过其他的 Java API 来间接使用它，如 J.U.C 包里面的整数原子类，其中的 compareAndSet()和 getAndIncrement()等方法都使用了 Unsafe 类的 CAS 操作。

我们不妨拿一段在上一章中没有解决的问题代码来看看如何使用 CAS 操作来避免阻

塞同步。

例如下面的一段代码。

```
public class VolatileTest {
  public static volatile int race=0 ;
  public static void increase() {
  race++;
  }
  private static final int THREADS COUNT=20;
  public  static void main (String[] args)  {
  Thread[]  threads=new Thread[THREADS COUNT];
  for (int i=0; i<THREADS—COUNT; i++)  {
  threads[i]  =new Thread(new Runnable()  {
  @Override
  public  void  run(){
  for (inti=o;  i(10000; i++)  {
  increase();
  }
  }
  })h
  threads [i].start()j
  }
  //等待所有累加线程都结束
  while  (Thread.activeCount()  >1)
  Thread.yield()j
  System.out.println (race);
  }
  }
```

上述代码发起了 20 个线程，每个线程对 race 变量进行 10000 次自增操作，如果这 段代码能够正确并发的话，最后输出的结果应该是 200000。读者运行完这段代码之后，并不会获得期望的结果，而且会发现每次运行程序，输出的结果都不一样，都是一个小于 200000 的数字，这是为什么呢？问题就出现在自增运算“race++”之中，我们用 Javap 反编译这段代码后会发现只有一行代码的 increase()方法在 Class 文件中是由 4 条字节码指令构成的(return 指令不是由 race++产生的，这条指令可以不算)，从字节码层面上很容易就分析出并发失败的原因了：当 getstatic 指令把 race 的值取到操作栈顶时，volatile 关键字保证了 race 的值在此时是正确的，但是在执行 iconst.1、iadd 这些指令的时候，其他线程可能已经把 race 的值加大了，而在操作栈顶的值就变成了过期的数据，所以 putstatic 指令执行后就可能把较小的 race 值同步回主内存之中。

如何才能让它具备原子性呢？把“race++”操作或 increase()方法用同步块包裹起来当然是一个办法，但是如果改成演示代码 17-4 的代码，那效率将会提高许多。

演示代码 17-4　Atomic 的原子自增运算

```
public class AtomicTest  {
public static AtomicInteger race-new AtomicInteger(0)j
public static void increase()  {
race .incrementAndGet();
}
private static final int THREADS COUNT 20;
```

```
public static void main(String[] args) throws Exception{
Thread[]  threads new Thread [THREADS COUNT];
for (int i=oj i(THREADS—COUNT; i++)  {
threads [i]  =new Thread(new Runnable()  {
@Override
public void run()  {
for (int i=o;i<10000; i++)  {
increase();
}
}
});
threads LiJ .start();
}
while (Thread.activeCount() >1)
Thread.yield();
System.out.println(race);
}
}
```

运行结果如下：

```
200000
```

使用 AtomicInteger 代替 int 后，程序输出了正确的结果，一切都要归功于 incrementAndGet()方法在一个无限循环中，不断尝试将一个比当前值大 1 的新值赋值给自己。如果失败了，那说明在执行“获取一设置”操作的时候值已经有了修改，于是再次循环进行下一次操作，直到设置成功为止。

尽管 CAS 看起来很美，但显然这种操作无法涵盖互斥同步的所有使用场景，并且 CAS 从语义上来说并不是完美的，存在这样的一个逻辑漏洞：如果一个变量 V 初次读取的时候是 A 值，并且在准备赋值的时候检查到它仍然为 A 值，那我们就能说它的值没有被其他线程改变过了吗？如果在这段期间它的值曾经被改成了 B，后来又被改回为 A，那 CAS 操作就会误认为它从来没有被改变过。这个漏洞称为 CAS 操作的“ABA”问题。J.U.C 包为了解决这个问题，提供了一个带有标记的原子引用类“AtomicStamped Reference”，它可以通过控制变量值的版本来保证 CAS 的正确性。不过目前来说这个类比较鸡肋，大部分情况下 ABA 问题不会影响程序并发的正确性，如果需要解决 ABA 问题，改用传统的互斥同步可能会比原子类更高效。

3. 无同步方案

要保证线程安全，并不是一定就要进行同步，两者没有因果关系。同步只是保障共享数据争用时的正确性的手段，如果一个方法本来就不涉及共享数据，那它自然就无需任何同步措施去保证正确性，因此会有一些代码天生就是线程安全的，笔者简单介绍其中的两类。

可重入代码(Reentrant Code)：这种代码也叫纯代码(Pure Code)，可以在代码执行的任何时刻中断它，转而去执行另外一段代码(包括递归调用它本身)，而在控制权返回后，原来的程序不会出现任何错误。相对线程安全来说，可重人性是更基本的特性，它可以保证

线程安全，即所有的可重入的代码都是线程安全的，但是并非所有的线程安全的代码都是可重入的。

可重入代码有一些共同的特征：例如不依赖存储在堆上的数据和公用的系统资源、用到的状态量都由参数中传入、不调用非可重入的方法等。我们可以通过一个简单一些的原则来判断代码是否具备可重入性：如果一个方法，它的返回结果是可以预测的，只要输入了相同的数据，就都能返回相同的结果，那它就满足可重入性的要求，当然也就是线程安全的。

线程本地存储(Thread Local Storage)：如果一段代码中所需要的数据必须与其他代码共享，那就看看这些共享数据的代码是否能保证在同一个线程中执行？如果能保证，我们就可以把共享数据的可见范围限制在同一个线程之内，这样，无需同步也能保证线程之间不出现数据争用的问题。

符合这种特点的应用并不少见，大部分使用消费队列的架构模式(如“生产者一消费者”模式)都会将产品的消费过程尽量在一个线程中消费完，其中最重要的一个应用实例就是经典 Web 交互模型中的“一个请求对应一个服务器线程”(Thread-per-Request)的处理方式，这种处理方式的广泛应用使得 Web 服务端的很多应用都可以使用线程本地存储来解决线程安全问题。

Java 语言中，如果一个变量要被多线程访问，可以使用 volatile 关键字声明它为“易变的”；如果一个变量要被某个线程独享，因为 Java 中没有类似 C++中_declspec(thread)这样的关键字，不过可以通过 java.lang.TbreadLocal 类来实现线程本地存储的功能。每一个线程的 Thread 对象中都有一个 ThreadLocalMap 对象，这个对象存储了一组以 ThreadLocal.threadLocalHashCode 为键，以本地线程变量为值的 K-V 值对，ThreadLocal 对象就是当前线程的 ThreadLocalMap 的访问入口，每一个 ThreadLocal 对象都包含了一个独一无二的 threadLocalHashCode 值，使用这个值就可以在线程 K-V 值对中找回对应的本地线程变量。

17.1.3 无状态类

线程安全的类一般是无状态的对象，或者类里面的变量是不可变的，下面举例说明什么是无状态类。

```
public class StatelessFactorizer implements Servlet {
    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        encodeIntoResponse(resp, factors);
    }
}
```

线程安全与线程模型有关系，因为每个线程都有自己的变量，并且变量放在自己的线程堆栈中(除了共享变量)，所以说无论多个线程怎么访问，该线程类都是安全。

也许有人会问是什么样的操作是原子性的呢？那么举个例子：

```
count++;
```

这个操作是原子性的吗？好像单道程序中该操作是原子性的，但在多线程中该操作不是原子性的；该操作分为读、修改、写入，因此该操作就会在多个线程中执行时会被切换，也就说明该操作会在执行过程中被切换给下一个线程。

看下面的代码：

```
public class UnsafeCountingFactorizer implements Servlet {
   private long count = 0;

   public long getCount() { return count; }

   public void service(ServletRequest req, ServletResponse resp) {
      BigInteger i = extractFromRequest(req);
      BigInteger[] factors = factor(i);
      ++count;
      encodeIntoResponse(resp, factors);
   }
}
```

上述例子有一个私有的但是多个线程共享的变量，而在 Service 方法中该方法体里面执行了++count，所以该类不是线程安全的。类失去原子性操作的另一个典型是：检查在运行，下面就为大家举个单例模式中的懒加载问题。

```
public class LazyInitRace {
   private ExpensiveObject instance = null;
   public ExpensiveObject getInstance() {
      if (instance == null)
         instance = new ExpensiveObject();
      return instance;
   }
}
```

大家可能会一下子知道这个没有什么问题，但是在多线程中运行该 getInstance 方法时，有可能两个或以上的线程会判断 instance 为空。所以说要尽量在判断和新建对象的时候不要被线程切换，也就是说保持操作的原子性的时候那么该类就是线程安全。

下面为大家阐明上面出现的两个问题的解决方案：

(1) 要想解决原子性的操作就需要给这些操作加锁。

(2) 要让每个线程知道，当有个线程修改对象或者修改变量时，其他线程都要可见，这个关系到 JVM 的类存模型。

首先第一个问题 read-modify-write 解决方案，我们引用了 JDK 包中的 java.lang.concurrent.atomic，该包里面可以让变量保持原子性操作，具体类的简单介绍如下所示。

- AtomicBoolean：可以用原子方式更新的 boolean 值。
- AtomicInteger：可以用原子方式更新的 int 值。
- AtomicIntegerArray：可以用原子方式更新其元素的 int 数组。
- AtomicIntegerFieldUpdater<T>：基于反射的实用工具，可以对指定类的指定 volatile int 字段进行原子更新。
- AtomicLong：可以用原子方式更新的 long 值。

- AtomicLongArray：可以用原子方式更新其元素的 long 数组。
- AtomicLongFieldUpdater<T>：基于反射的实用工具，可以对指定类的指定 volatile long 字段进行原子更新。
- AtomicMarkableReference<V>：AtomicMarkableReference 维护带有标记位的对象引用，可以原子方式对其进行更新。
- AtomicReference<V>：可以用原子方式更新的对象引用。
- AtomicReferenceArray<E>：可以用原子方式更新其元素的对象引用数组。
- AtomicReferenceFieldUpdater<T,V>：基于反射的实用工具，可以对指定类的指定 volatile 字段进行原子更新。
- AtomicStampedReference<V>：AtomicStampedReference 维护带有整数“标志”的对象引用，可以用原子方式对其进行更新。

例如下面的代码：

```
public class CountingFactorizer implements Servlet {
    private final AtomicLong count = new AtomicLong(0);

    public long getCount() { return count.get(); }

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        count.incrementAndGet();
        encodeIntoResponse(resp, factors);
    }
}
```

第二个问题解决方案就是加上关键字 sychronized。

17.2 锁 优 化

高效并发是 JDK 1.6 的一个重要主题，HotSpot 虚拟机开发团队在这个版本上花费了大量的精力去实现各种锁优化技术，如适应性自旋(Adaptive Spinning)、锁消除(Lock Elimination)、锁粗化(Lock Coarsening)、轻量级锁(Lightweight Locking)、偏向锁 (Biased Locking)等，这些技术都是为了在线程之间更高效地共享数据，以及解决竞争问题，从而提高程序的执行效率。

17.2.1 自旋锁与自适应自旋

前面我们讨论互斥同步的时候，提到了互斥同步对性能最大的影响是阻塞的实现，挂起线程和恢复线程的操作都需要转入内核态中完成，这些操作给系统的并发性能带来了很大的压力。同时，虚拟机的开发团队也注意到在许多应用上，共享数据的锁定状态只会持续很短的一段时间，为了这段时间去挂起和恢复线程并不值得。如果物理机器有一个以上的处理器，能让两个或以上的线程同时并行执行，我们就可以让后面请求锁的那个线程

“稍等一会儿”，但不放弃处理器的执行时间，看看持有锁的线程是否很快就会释放锁。为了让线程等待，我们只需让线程执行一个忙循环(自旋)，这项技术就是所谓的自旋锁。

自旋锁在 JDK 1.4.2 中就已经引入，只不过默认是关闭的，可以使用-XX:+UseSpinning 参数来开启，在 JDK 1.6 中就已经改为默认开启了。自旋等待不能代替阻塞，且先不说对处理器数量的要求，自旋等待本身虽然避免了线程切换的开销，但它是要占用处理器时间的，所以如果锁被占用的时间很短，自旋等待的效果就会非常好，反之如果锁被占用的时间很长，那么自旋的线程只会白白消耗处理器资源，而不会做任何有用的工作，反而会带来性能的浪费。因此自旋等待的时间必须要有一定的限度，如果自旋超过了限定的次数仍然没有成功获得锁，就应当使用传统的方式去挂起线程了。自旋次数的默认值是 10 次，用户可以使用参数-XX:PreBlockSpin 来更改。

在 JDK 1.6 中引入了自适应的自旋锁。自适应意味着自旋的时间不再固定了，而是由前一次在同一个锁上的自旋时间及锁的拥有者的状态来决定。如果在同一个锁对象上，自旋等待刚刚成功获得过锁，并且持有锁的线程正在运行中，那么虚拟机就会认为这次自旋也很有可能再次成功，进而它将允许自旋等待持续相对更长的时间，比如 100 个循环。另一方面，如果对于某个锁，自旋很少成功获得过，那在以后要获取这个锁时将可能省略掉自旋过程，以避免浪费处理器资源。有了自适应自旋，随着程序运行和性能监控信息的不断完善，虚拟机对程序锁的状况预测就会越来越准确，虚拟机就会变得越来越“聪明”了。

17.2.2 锁消除

锁消除是指虚拟机即时编译器在运行时，对一些代码上要求同步，但是被检测到不可能存在共享数据竞争的锁进行消除。锁消除的主要判定依据来源于逃逸分析的数据支持，如果判断到一段代码中，在堆上的所有数据都不会逃逸出去被其他线程访问到，那就可以把它们当做栈上数据对待，认为它们是线程私有的，同步加锁自然就无需进行。

也许读者会有疑问，变量是否逃逸，对于虚拟机来说需要使用数据流分析来确定，但是程序员自己应该是很清楚的，怎么会在明知道不存在数据争用的情况下要求同步呢？答案是有许多同步措施并不是程序员自己加入的，同步的代码在 Java 程序中的普遍程度也许超过了大部分读者的想象。我们来看看下面演示代码 17-5 中的例子，这段非常简单的代码仅仅是输出三个字符串相加的结果，无论是源码字面上还是程序语义上都没有同步。

演示代码 17-5

```
public  String concatString(String s1, String s2, String s3) {
    return  s1 + s2 + s3;
}
```

由于 String 是一个不可变的类，对字符串的连接操作总是通过生成新的 String 对象来进行的，因此 Javac 编译器会对 String 连接做自动优化。在 JDK 1.5 之前，会转化为 StringBuffer 对象的连续 append()操作，在 JDK 1.5 及以后的版本中，会转化为 StringBuilder 对象的连续 append()操作。即演示代码 17-5 中的代码可能会变成演示代码 17-6 的样子。

演示代码 17-6

```
public String concatString(String s1, String s2, String s3) {
    StringBuffer sb = new StringBuffer();
    sb.append(s1);
    sb.append(s2);
    sb.append(s3);
    return sb.toString();
}
```

现在大家还认为这段代码没有涉及同步吗？每个 StringBuffer.append()方法中都有一个同步块，锁就是 sb 对象。虚拟机观察变量 sb，很快就会发现它的动态作用域被限制在 concatString()方法的内部。也就是 sb 的所有引用永远不会“逃逸”到 concatString()方法之外，其他线程无法访问到它，所以这里虽然有锁，但是可以被安全地消除掉，在即时编译之后，这段代码就会忽略掉所有的同步而直接执行了。

17.2.3 锁膨胀

原则上，我们在编写代码的时候，总是推荐将同步块的作用范围限制得尽量小——只在共享数据的实际作用域中才进行同步，这样是为了使得需要同步的操作数量尽可能变小，如果存在锁竞争，那等待锁的线程也能尽快地拿到锁。

大部分情况下，上面的原则都是正确的，但是如果一系列的连续操作都对同一个对象反复加锁和解锁，甚至加锁操作是出现在循环体中的，那即使没有线程竞争，频繁地进行互斥同步操作也会导致不必要的性能损耗。演示代码 17-6 中连续的 append()方法就属于这类情况。如果虚拟机探测到有这样一串零碎的操作都对同一个对象加锁，将会把加锁同步的范围扩展(粗化)到整个操作序列的外部，以演示代码 17-6 为例，就是扩展到第一个 append()操作之前直至最后一个 append()操作之后，这样只需要加锁一次就可以了。

17.2.4 轻量级锁

轻量级锁是 JDK 1.6 中加入的新型锁机制，它名字中的“轻量级”是相对于使用操作系统互斥量来实现的传统锁而言的，因此传统的锁机制就被称为“重量级”锁。首先需要强调一点的是，轻量级锁并不是用来代替重量级锁的，它的本意是在没有多线程竞争的前提下，减少传统的重量级锁使用操作系统互斥量产生的性能消耗。

要理解轻量级锁，以及后面会讲到的偏向锁的原理和运作过程，必须从 HotSpot 虚拟机的对象(对象头部分)的内存布局开始介绍。HotSpot 虚拟机的对象头(Object Header)分为两部分信息，第一部分用于存储对象自身的运行时数据，如哈希码(HashCode)、GC 分代年龄(Generational GC Age)等，这部分数据的长度在 32 位和 64 位的虚拟机中分别为 32 个和 64 个 Bits，官方称它为“Mark Word”，它是实现轻量级锁和偏向锁的关键。另外一部分用于存储指向方法区对象类型数据的指针，如果是数组对象的话，还会有一个额外的部分用于存储数组长度。

对象头信息是与对象自身定义的数据无关的额外存储成本，考虑到虚拟机的空间效率，Mark Word 被设计成一个非固定的数据结构以便在极小的空间内存储尽量多的信息，

它会根据对象的状态复用自己的存储空间。例如在 32 位的 HotSpot 虚拟机中对象未被锁定的状态下，Mark Word 的 32 个 Bits 空间中的 25Bits 用于存储对象哈希码(HashCode)，4Bits 用于存储对象分代年龄，2Bits 用于存储锁标志位，1Bit 固定为 0，在其他状态(轻量级锁定、重量级锁定、GC 标记、可偏向)下对象的存储内容如表 17-1 所示。

表 17-1　HotSpot 虚拟机对象头 Mark Word

存储内容	标 志 位	状　态
对象哈希码、对象分代年龄	01	未锁定
指向锁记录的指针	00	轻量级锁定
指向重量级锁的指针	10	膨胀(重量级锁定)
空，不需要记录信息	11	GC 标记
偏向线程 ID、偏向时间戳、对象分代年龄	01	可偏向

简单地介绍完了对象的内存布局，我们把话题返回到轻量级锁的执行过程上。在代码进入同步块的时候，如果此同步对象没有被锁定(锁标志位为“01”状态)，虚拟机首先将在当前线程的栈帧中建立一个名为锁记录(Lock Record)的空间，用于存储锁对象目前的 Mark Word 的拷贝(官方把这份拷贝加了一个 Displaced 前缀，即 Displaced Mark Word)，这时候线程堆栈与对象头的状态如图 17-1 所示。然后，虚拟机将使用 CAS 操作尝试将对象的 Mark Word 更新为指向 Lock Record 的指针。如果这个更新动作成功了，那么这个线程就拥有了该对象的锁，并且对象 Mark Word 的锁标志位(Mark Word 的最后两个 Bits)将转变为“00”，即表示此对象处于轻量级锁定的状态，这时候线程堆栈与对象头的状态如图 17-2 所示。

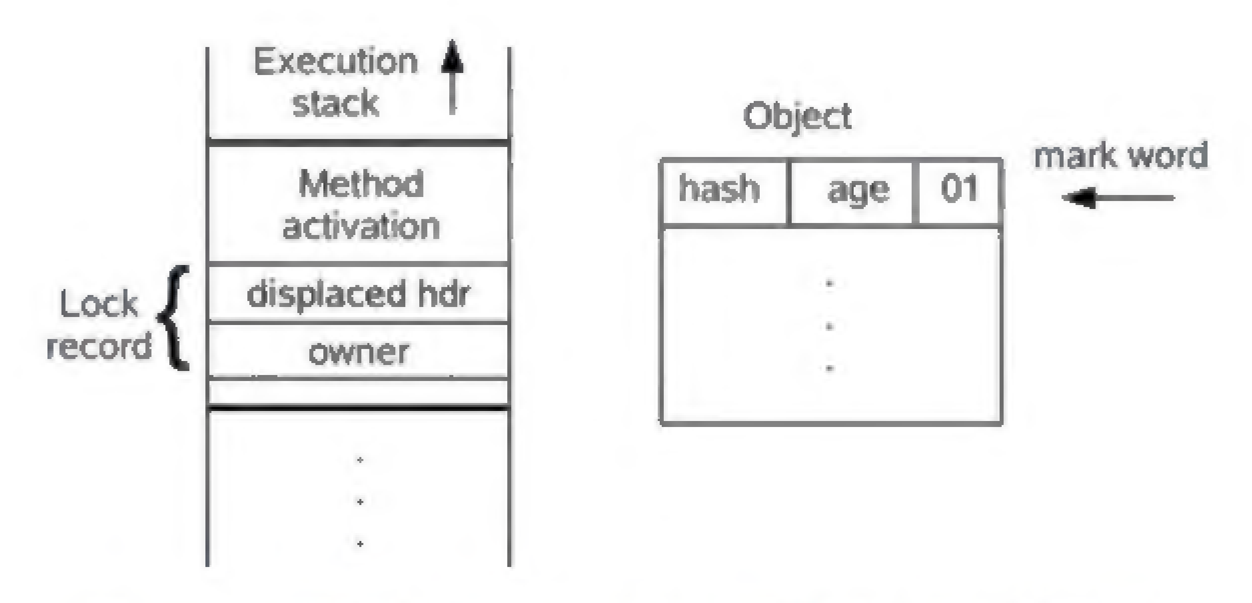


图 17-1　轻量级锁 CAS 操作之前堆栈与对象的状态

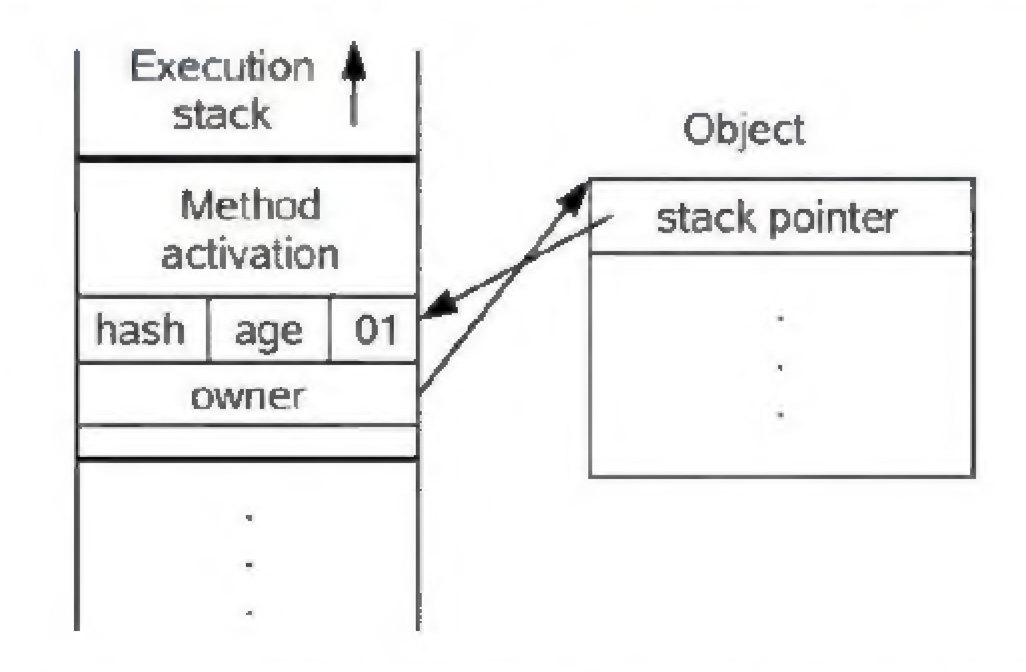


图 17-2　轻量级锁 CAS 操作之后堆栈与对象的状态

注意：上述图 17-1 和图 17-2 来源于大师 Paul Hohensee 所写的 PPT《The Hotspot Java Virtual Machine》。

如果这个更新操作失败了，虚拟机首先会检查对象的 Mark Word 是否指向当前线程的栈帧，如果是就说明当前线程已经拥有了这个对象的锁，可以直接进入同步块继续执行，否则说明这个锁对象已经被其他线程抢占了。如果有两条以上的线程争用同一个锁，那轻量级锁就不再有效，要膨胀为重量级锁，锁标志的状态值变为 10，Mark Word 中存储的就是指向重量级锁(互斥量)的指针，后面等待锁的线程也要进入阻塞状态。

上面描述的是轻量级锁的加锁过程，它的解锁过程也是通过 CAS 操作来进行的，如果对象的 Mark Word 仍然指向着线程的锁记录，那就用 CAS 操作把对象当前的 MarkWord 和线程中复制的 Displaced Mark Word 替换回来，如果替换成功，整个同步过程就完成了。如果替换失败，说明有其他线程尝试过获取该锁，那就要在释放锁的同时，唤醒被挂起的线程。

轻量级锁能提升程序同步性能的依据是“对于绝大部分的锁，在整个同步周期内都是不存在竞争的”，这是一个经验数据。如果没有竞争，轻量级锁使用 CAS 操作避免了使用互斥量的开销，但如果存在锁竞争，除了互斥量的开销外，还额外发生了 CAS 操作，因此在有竞争的情况下，轻量级锁会比传统的重量级锁更慢。

17.2.5 偏向锁

偏向锁也是 JDK 1.6 中引入的一项锁优化，它的目的是消除数据在无竞争情况下的同步原语，进一步提高程序的运行性能。如果说轻量级锁是在无竞争的情况下使用 CAS 操作去消除同步使用的互斥量，那偏向锁就是在无竞争的情况下把整个同步都消除掉，连 CAS 操作都不做了。

偏向锁的“偏”，就是偏心的“偏”、偏袒的“偏”。它的意思是这个锁会偏向于第一个获得它的线程，如果在接下来的执行过程中，该锁没有被其他的线程获取，则持有偏向锁的线程将永远不需要再进行同步。

如果读者读懂了前面轻量级锁中关于对象头 Mark Word 与线程之间的操作过程，那偏向锁的原理理解起来就会很简单。假设当前虚拟机启用了偏向锁(启用参数-XX:+UseBiasedLocking，这是 JDK 1.6 的默认值)，那么，当锁对象第一次被线程获取的时候，虚拟机将会把对象头中的标志位设为“01”，即偏向模式。同时使用 CAS 操作把获取到这个锁的线程的 ID 记录在对象的 Mark Word 之中，如果 CAS 操作成功，持有偏向锁的线程以后每次进入这个锁相关的同步块时，虚拟机都可以不再进行任何同步操作，例如 Locking、Unlocking 及对 Mark Word 的 Update 等。

当有另外一个线程去尝试获取这个锁时，偏向模式就宣告结束。根据锁对象目前是否处于被锁定的状态，撤销偏向(Revoke Bias)后恢复到未锁定(标志位为“01”)或轻量级锁定(标志位为“00”)的状态，后续的同步操作就如上面介绍的轻量级锁那样执行。偏向锁、轻量级锁的状态转化及对象 Mark Word 的关系如图 17-3 所示。

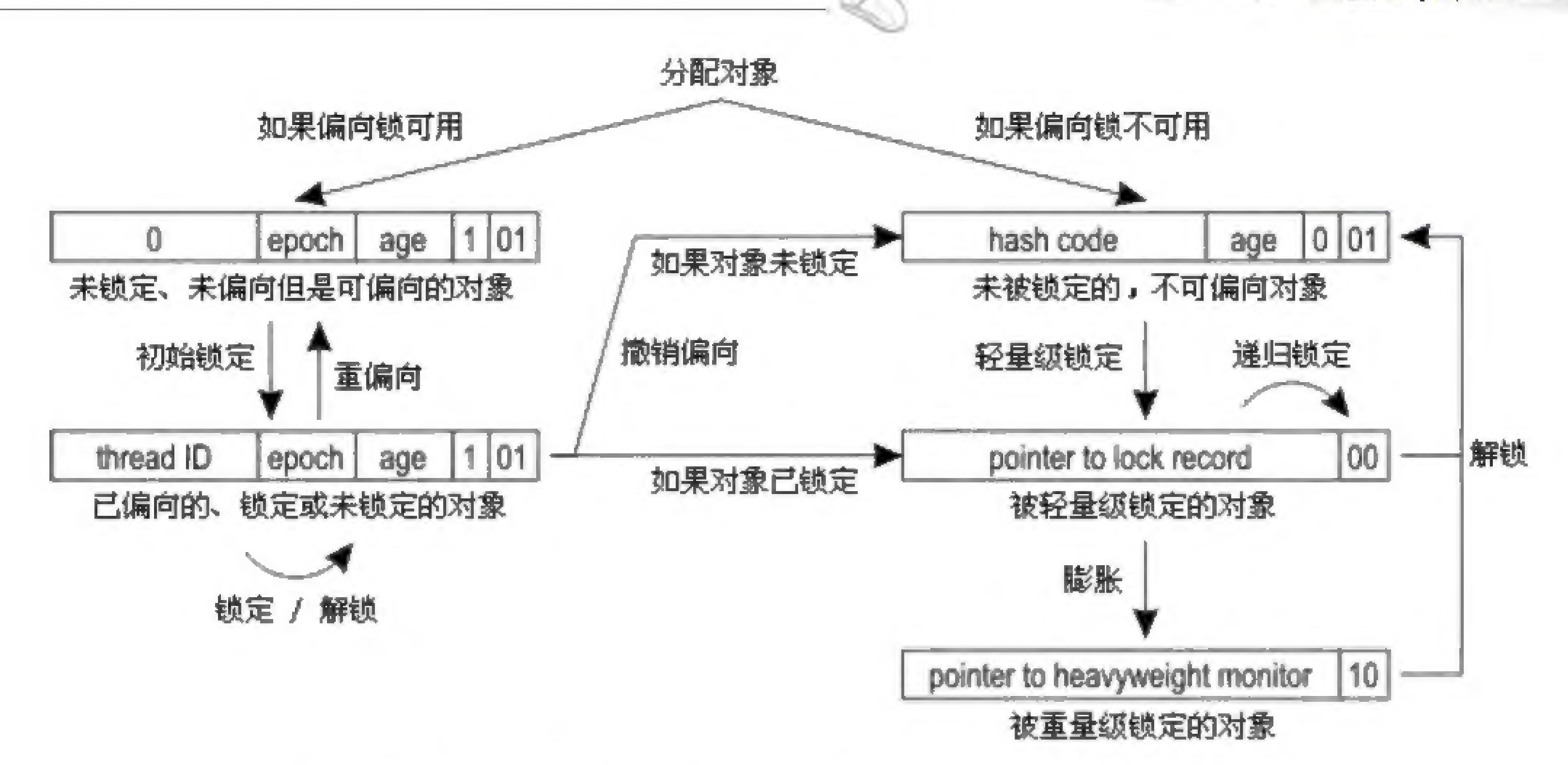


图 17-3　偏向锁、轻量级锁的状态转化及对象 Mark Word 的关系

偏向锁可以提高带有同步但无竞争的程序性能。它同样是一个带有效益权衡(Trade Off)性质的优化，也就是说它并不一定总是对程序运行有利，如果程序中大多数的锁都总是被多个不同的线程访问，那偏向模式就是多余的。在具体问题具体分析的前提下，有时候使用参数-XX:-UseBiasedLocking 来禁止偏向锁优化反而可以提升性能。

在 JDK6 中，偏向锁是默认启用的。它提高了单线程访问同步资源的性能。但试想一下，如果你的同步资源或代码一直都是多线程访问的，那么消除偏向锁这一步骤对你来说就是多余的。事实上，消除偏向锁的开销还是蛮大的。所以在你非常熟悉自己代码的前提下，建议禁用偏向锁 -XX:-UseBiasedLocking。